



# Netty进阶之路

## 跟着案例学Netty

李林锋 著



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>





版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



夕阳半卧楼宇间，  
秋风送凉白露天。  
谁言序猿欠风雅，  
架构雕琢美名传。



## 李林锋 ▶

◆10年Java NIO通信框架、平台中间件架构设计和开发经验。

◆目前在华为终端应用市场负责业务微服务化、云化、全球化等相关设计和开发工作。

◆《Netty权威指南》和《分布式服务框架原理与实践》作者。

微信公众号：Netty之家





# Netty进阶之路

## 跟着案例学Netty

李林锋 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING



## 内 容 简 介

Netty 将 Java NIO 接口封装, 提供了全异步编程方式, 是各大 Java 项目的网络应用开发必备神器。本书作者是国内 Netty 技术的先行者和布道者, 本书是他继《Netty 权威指南》之后的又一力作。

在本书中, 作者将在过去几年实践中遇到的问题, 以及 Netty 学习者咨询的相关问题, 进行了归纳和总结, 以问题案例做牵引, 通过对案例进行剖析, 讲解问题背后的原理, 并结合 Netty 源码分析, 让读者能够真正掌握 Netty, 在实际工作中少犯错。

本书中的案例涵盖了 Netty 的启动和停止、内存、并发多线程、性能、可靠性、安全等方面, 囊括了 Netty 绝大多数常用的功能及容易让人犯错的地方。在案例的分析过程中, 还穿插讲解了 Netty 的问题定位思路、方法、技巧, 以及解决问题使用的相关工具, 对读者在实际工作中用好 Netty 具有很大的帮助和启发作用。

本书适合架构师、设计师、开发工程师、测试工程师, 以及对 Java NIO 框架、Netty 感兴趣的其他相关人士阅读。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有, 侵权必究。

### 图书在版编目(CIP)数据

Netty 进阶之路: 跟着案例学 Netty / 李林锋著. —北京: 电子工业出版社, 2019.1  
ISBN 978-7-121-35262-1

I. ①N… II. ①李… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2018)第 240985 号

责任编辑: 董 英

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 21.25 字数: 425 千字

版 次: 2019 年 1 月第 1 版

印 次: 2019 年 1 月第 1 次印刷

定 价: 79.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: 010-51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。





# 序 1

随着近几年的快速发展，移动互联网系统的复杂度快速上升。为了满足业务快速迭代的需求，同时提高系统的可靠性和可维护性，越来越多的大型系统后台开始采用微服务架构。以华为应用市场为例，目前后台微服务数量达上百个，服务器多达数千台。服务器数量的快速增加，必然导致后台系统复杂度的快速增加，从而推动后台技术架构的持续演进。

在单体系统中，一个请求通常只涉及一个部件。但微服务化后，一个请求可能涉及数个微服务的配合。传统的后台服务通常采用同步阻塞调用方式，一个请求独占一个线程，只有在请求处理完成后，线程才能被释放。如果一个请求涉及多个微服务之间的调用，那么每个微服务都需要一个专门的连接和线程来处理这个请求。而系统的线程数和连接数都是有限的。当系统调用关系越来越复杂，一个很小的问题（如网络抖动、突发请求或 GC 引起的延时增加），都可能导致线程阻塞，引发雪崩，导致整个系统不可用。另一方面，同步阻塞的方式很容易导致系统的资源利用率降低。因此，业界领先的互联网公司，都已经或者正在切换到全栈异步方式。

相比同步 I/O 简单直接的编程模型，异步 I/O 在编程模型上有较大的差异，对开发人员也有更高的要求，同时问题的定位也更为复杂。Netty 是当前业界应用最广泛的 Java 开源异步框架。Netty 框架能显著降低异步开发的门槛，使开发人员聚焦业务逻辑，免于处理复杂的底层通信机制和线程模型，从而能够简单和快速地开发异步应用。时至今日，越来越多的国内公司开始使用 Netty 来构建应用，使用 Netty 的开发者也日益增加。

随着 Netty 应用的不断深入，大家在使用过程中也会遇到各种各样的问题。相比 Netty 的火热，市场上 Netty 相关的书籍却很少。作者几年前出版的著作《Netty 权威指南》是国

## Netty 进阶之路：跟着案例学 Netty

内第一本系统化讲解 Netty 原理和架构的书籍，在市场上取得了良好的反响。《Netty 进阶之路：跟着案例学 Netty》是作者在 Netty 方面的又一力作。该书从一个个典型的问题出发，让读者能够带着问题来展开学习，并通过代码解读、原理分析和问题总结，对每个问题抽丝剥茧地深入解析；同时，能够通过一个问题，将相关领域的知识理解透彻，达到举一反三的效果，进而实现对 Netty 的系统性学习与理解。

李林锋在电信软件行业有着近十年的异步和服务化方面的开发和架构设计经验，是华为公司该领域的专家。他从 2017 年开始负责华为应用市场在异步和微服务化方面的工作，主导了华为应用市场后台的异步化和微服务化演进，将电信软件严谨、稳定、高性能的优势，与移动互联网海量用户、高并发场景结合起来，显著提升了华为应用市场的可用性和性能。本书凝结了作者多年来在异步化工作方面的经验，将成为希望精通 Netty 开发的读者的重要参考书。

华为应用市场总架构师 刘连喜





## 序 2

随着互联网对各行各业的渗透，“连接”便成了我们看到的所有美好应用背后的基石。人、系统、物三者之间无所不在的连接让我们甚至感觉这世界已经成为了一个整体。细数一些计算机领域的热门技术，例如云计算、微服务、物联网等，其背后的核心还是连接。在这样一个背景下，掌握 Netty 可以算得上是一个开发人员最重要的技能。

就像码农的学名其实是计算机工程师一样，计算机其实是一个工程学科，是一个天生看重实践的学科。在现在的环境里，各种技术文档在网上一抓一大把，但是仍然有一些东西很难得到，那就是你需要填的那些坑。

李林锋长期工作在技术一线，构建的是压力最大的系统，保证的是要求最高的服务，解决的是诡异难解的问题。而难能可贵的是，他把他自己填过的那些坑都详尽地记录了下来，不仅提供思路，还分析原理，让所有问题都无所遁形。

《Netty 进阶之路：跟着案例学 Netty》就是能够助人出坑的干货。

佛教的修行次第是“信，解，行，证”。李林锋一直以来对技术的热情、“知行合一”的行事方式和对问题刨根问底的工作态度非常难得，相信通过这本书也可以让你由“解”入“行”，他趟过那些大坑的经验也一定会助你早日在 Netty 学习的路上实现“自证”。

华为云高级架构师 张琦



# 前言

自 2014 年《Netty 权威指南》出版后，我在技术网站上相继写了一些 Netty 专题文章，涵盖性能、线程模型、安全性等知识点，受到很多读者的喜爱。在 4 年多里，很多读者及 Netty 学习者向我咨询 Netty 相关的问题，这些问题加起来多达上千个，通过对问题做汇总和分析，可以归纳为如下几类：

(1) Netty 初学者，想了解学习 Netty 需要储备哪些技能，掌握哪些知识点，有什么学习技巧可以更快地掌握 Netty。

(2)《Netty 权威指南》的读者，学习完理论知识后，想在实际项目中使用，但是真正跟具体项目结合在一起解决实际问题时，又感觉比较棘手，不知道自己使用的方式是否是最优的，希望能够多学一些案例实践方面的知识，以便更好地在业务中使用 Netty。

(3) 在实际项目中遇到了问题的工程师，由于对 Netty 底层细节掌握得不扎实，无法有效地定位并解决问题。

Netty 的一个特点就是入门相对容易，但是真正掌握并精通是非常困难的，原因有如下几个：

(1) 涉及的知识面比较广。Netty 作为一个高性能的 NIO 通信框架，涉及的知识点包括网络通信、多线程编程、序列化和反序列化、异步和同步、SSL/TLS 安全、内存池、HTTP 等各种协议栈，这些知识点在 Java 语言中本身就是难点和重点，如果对这些基础知识掌握不扎实，是很难真正掌握好 Netty 的。

(2) 调试比较困难。因为大量使用异步编程接口，以及消息处理过程中的各种线程切换，相比传统同步代码，Netty 代码调试难度比较大。





(3) 类继承层次比较深, 有些代码很晦涩 (例如内存池)。对于初学者而言, 通过阅读代码来掌握 Netty 的难度还是很大的。

(4) 代码规模庞大。目前, Netty 的代码规模已经非常庞大, 特别是协议栈部分, 提供了对 HTTP/2、MQTT、WebSocket 等各种协议的支持, 相关代码非常多。如果学习方式不当, 抓不住重点, 则全量阅读 Netty 源码, 既耗时又很难吃透, 很容易半途而废。

(5) 资料零散, 缺乏与实践相关的案例。网上 Netty 的各种资料非常多, 但是都以理论讲解为主, Netty 在各行业中的应用、问题定位技巧及案例实践方面的资料很少, 缺乏系统性的实践总结, 是 Netty 学习的一大痛点。

在过去的几年中, 我利用业余时间尽量帮大家答疑解惑, 但实际上一个人很难回答所有读者的问题, 有些问题需要业务描述、故障场景、日志, 甚至要看源码, 而且需要反复多次沟通来弄清楚问题, 对于个人而言, 时间和精力都很难得到保证。另外, 一些比较常见的问题, 例如服务端接收不到客户端的消息, 定位手段是可以固化下来的。很多读者也希望我能写一本 Netty 实践和案例方面的书, 通过案例讲解让大家更好地在项目中使用 Netty, 解决遇到的实际问题。

于是我对手头大家咨询的问题做了归类分析, 结合我们自己的业务和平台多年来在 Netty 实践中积累的经验, 写作了本书。本书以问题案例做牵引, 通过对案例进行剖析, 讲解问题背后的原理, 并结合 Netty 源码分析, 让读者能够真正掌握 Netty, 在实际工作中少犯错。在案例的分析过程中, 还穿插讲解了 Netty 的问题定位思路、方法、技巧, 以及解决问题使用的相关工具, “授人以鱼不如授人以渔”, 只有掌握了这些才能更放心地使用 Netty。

本书的内容分类主要包括:

- (1) Netty 的启动和停止
- (2) Netty 的内存
- (3) Netty 的并发多线程
- (4) Netty 的性能
- (5) Netty 的可靠性
- (6) Netty 的安全





## （7）Netty 的实践

## （8）Netty 的学习

书中的案例涵盖了 Netty 绝大多数常用的功能，以及容易犯错的地方，具有通用性和普遍性。学习这些案例，对于在实际业务工作中用好 Netty 有很大的帮助和启发作用。另外，在讲解 Netty 框架本身的同时，也会穿插一些背景知识介绍，例如 Java 信号量和优雅停机机制、Java 的 NIO 类库、HTTP 协议栈等。知识都是相互关联的，很难在基础知识不扎实的情况下掌握更高阶的知识。

通过本书的学习，希望广大 Netty 初学者和爱好者能够更快、更好地进入高级阶段，在项目中用好 Netty，为业务创造更多的价值。

尽管我也有技术洁癖，希望诸事完美，但是由于 Netty 代码的庞杂和涉及的知识点太多，以及受限于我个人的经历和水平，很难在一本书里同时满足所有读者的诉求。本书如有遗漏或者错误，恳请大家及时批评和指正，如果大家有好的建议或者想法，也可以联系我。联系方式如下：

◎ 微信：Nettying

◎ 新浪微博：Nettying

能够完成本书要感谢很多人，首先感谢华为公司给我提供了足够大的舞台，感谢华为消费者云服务应用市场团队领导张凡、叶文武、刘连喜等，以及这些年与我在平台和业务团队一起战斗过的架构师、设计师、开发工程师、测试工程师和资料员等同事。

其次感谢我的家人，你们一直在背后默默地支持我。感谢参与本书编辑的英姐、美工及其他人员，你们的辛苦换来了本书的如期上市。

最后感谢所有《Netty 权威指南》和《分布式服务框架原理与实践》的读者，你们的支持和鼓励是我写作本书的动力源泉。

李林锋

2018 年国庆节于南京



## 读者服务

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），您即可享受以下服务。

- ◎ **下载资源：**本书提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- ◎ **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- ◎ **与作者交流：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/35262>





# 目 录

第 1 章	Netty 服务端意外退出案例 .....	1
1.1	Netty 服务端意外退出问题 .....	1
1.1.1	Java Daemon 线程简介 .....	2
1.1.2	Netty 服务端启动原理 .....	4
1.1.3	如何防止 Netty 服务端意外退出 .....	6
1.1.4	实际项目中的优化策略 .....	8
1.2	Netty 优雅退出机制 .....	9
1.2.1	Java 优雅退出机制 .....	10
1.2.2	Java 优雅退出的注意点 .....	12
1.2.3	Netty 优雅退出机制 .....	14
1.2.4	Netty 优雅退出原理和源码分析 .....	15
1.2.5	Netty 优雅退出的一些误区 .....	20
1.3	总结 .....	21
第 2 章	Netty 客户端连接池资源泄漏案例 .....	22
2.1	Netty 连接池资源泄漏问题 .....	22
2.1.1	连接池创建代码 .....	23
2.1.2	内存溢出和线程膨胀 .....	23
2.1.3	错用 NIO 编程模式 .....	25
2.1.4	正确的连接池创建方式 .....	26
2.1.5	并发安全和资源释放 .....	28
2.2	Netty 客户端创建机制 .....	29
2.2.1	Java NIO 客户端创建原理分析 .....	29

2.2.2	Netty 客户端创建原理分析 .....	32
2.2.3	Bootstrap 工具类源码分析 .....	34
2.3	总结 .....	36
第 3 章	Netty 内存池泄漏疑云案例 .....	37
3.1	Netty 内存池泄漏问题 .....	37
3.1.1	路由转发服务代码 .....	38
3.1.2	响应消息内存释放玄机 .....	39
3.1.3	采集堆内存快照分析 .....	42
3.1.4	ByteBuf 申请和释放的理解误区 .....	45
3.2	Netty 内存池工作机制 .....	48
3.2.1	内存池的性能优势 .....	48
3.2.2	内存池工作原理分析 .....	51
3.2.3	内存池核心代码分析 .....	54
3.3	总结 .....	58
第 4 章	ByteBuf 故障排查案例 .....	59
4.1	HTTP 协议栈 ByteBuf 使用问题 .....	59
4.1.1	HTTP 响应 Body 获取异常 .....	59
4.1.2	ByteBuf 非法引用问题 .....	63
4.1.3	ByteBuf 使用注意事项 .....	66
4.2	Netty ByteBuf 实现机制 .....	67
4.2.1	Java 原生 ByteBuffer 的局限性 .....	67
4.2.2	Netty ByteBuf 工作原理分析 .....	67
4.2.3	ByteBuf 引用计数器工作原理和源码分析 .....	70
4.3	总结 .....	73
第 5 章	Netty 发送队列积压导致内存泄漏案例 .....	74
5.1	Netty 发送队列积压案例 .....	74
5.1.1	高并发故障场景 .....	74
5.1.2	内存泄漏原因分析 .....	76
5.1.3	如何防止发送队列积压 .....	78
5.1.4	其他可能导致发送队列积压的因素 .....	80

5.2	Netty 消息发送工作机制 .....	82
5.2.1	WriteAndFlushTask 原理和源码分析 .....	83
5.2.2	ChannelOutboundBuffer 原理和源码分析 .....	86
5.2.3	消息发送源码分析 .....	88
5.2.4	消息发送高低水位控制 .....	94
5.3	总结 .....	95
第 6 章	API 网关高并发压测性能波动案例 .....	96
6.1	高并发压测性能波动问题 .....	96
6.1.1	故障场景模拟 .....	96
6.1.2	性能波动原因定位 .....	98
6.1.3	主动内存泄漏定位法 .....	101
6.1.4	网关类产品的优化建议 .....	102
6.2	Netty 消息接入内存申请机制 .....	102
6.2.1	消息接入的内存分配原理和源码分析 .....	102
6.2.2	Netty ByteBuf 的动态扩容原理和源码分析 .....	107
6.3	总结 .....	108
第 7 章	Netty ChannelHandler 并发安全案例 .....	109
7.1	Netty ChannelHandler 并发安全问题 .....	109
7.1.1	串行执行的 ChannelHandler .....	110
7.1.2	跨链路共享的 ChannelHandler .....	114
7.1.3	ChannelHandler 的并发陷阱 .....	116
7.2	Netty ChannelHandler 工作机制 .....	118
7.2.1	职责链 ChannelPipeline 原理和源码分析 .....	118
7.2.2	用户自定义 Event 原理和源码分析 .....	122
7.3	总结 .....	123
第 8 章	车联网服务端接收不到车载终端消息案例 .....	124
8.1	车联网服务端接收不到车载终端消息问题 .....	124
8.1.1	故障现象 .....	125
8.1.2	故障期线程堆栈快照分析 .....	126
8.1.3	NioEventLoop 线程防挂死策略 .....	128

8.2	NioEventLoop 线程工作机制 .....	129
8.2.1	I/O 读写操作原理和源码分析 .....	130
8.2.2	异步任务执行原理和源码分析 .....	133
8.2.3	定时任务执行原理和源码分析 .....	135
8.2.4	Netty 多线程最佳实践 .....	137
8.3	总结 .....	137
第 9 章	Netty 3.X 版本升级案例 .....	139
9.1	Netty 3.X 的版本升级背景 .....	139
9.1.1	被迫升级场景 .....	140
9.1.2	升级不当遭遇各种问题 .....	140
9.2	版本升级后数据被篡改问题 .....	141
9.2.1	数据篡改原因分析 .....	142
9.2.2	问题总结 .....	143
9.3	升级后上下文丢失问题 .....	143
9.3.1	上下文丢失原因分析 .....	144
9.3.2	依赖第三方线程模型的思考 .....	144
9.4	升级后应用遭遇性能下降问题 .....	145
9.4.1	性能下降原因分析 .....	145
9.4.2	性能优化建议 .....	146
9.5	Netty 线程模型变更分析 .....	147
9.5.1	Netty 3.X 版本线程模型 .....	147
9.5.2	Netty 4.X 版本线程模型 .....	149
9.5.3	线程模型变化点源码分析 .....	150
9.5.4	线程模型变化总结 .....	152
9.6	总结 .....	154
第 10 章	Netty 并发失效导致性能下降案例 .....	155
10.1	业务 ChannelHandler 无法并发执行问题 .....	155
10.1.1	服务端并发设计相关代码分析 .....	155
10.1.2	无法并行执行的 EventExecutorGroup .....	159
10.1.3	并行执行优化策略和结果 .....	161
10.2	Netty DefaultEventExecutor 工作机制 .....	163

10.2.1	DefaultEventExecutor 原理和源码分析 .....	164
10.2.2	业务线程池优化策略 .....	165
10.2.3	Netty 线程绑定机制原理和源码分析 .....	168
10.3	总结 .....	170
<b>第 11 章</b>	<b>IoT 百万长连接性能调优案例 .....</b>	<b>171</b>
11.1	海量长连接接入面临的挑战 .....	171
11.1.1	IoT 设备接入特点 .....	172
11.1.2	IoT 服务端性能优化场景 .....	172
11.1.3	服务端面临的性能挑战 .....	172
11.2	智能家居内存泄漏问题 .....	173
11.2.1	服务端内存泄漏原因定位 .....	173
11.2.2	问题背后的一些思考 .....	174
11.3	操作系统参数调优 .....	174
11.3.1	文件描述符 .....	175
11.3.2	TCP/IP 相关参数 .....	175
11.3.3	多网卡队列和软中断 .....	177
11.4	Netty 性能调优 .....	177
11.4.1	设置合理的线程数 .....	177
11.4.2	心跳优化 .....	180
11.4.3	接收和发送缓冲区调优 .....	183
11.4.4	合理使用内存池 .....	184
11.4.5	防止 I/O 线程被意外阻塞 .....	185
11.4.6	I/O 线程和业务线程分离 .....	187
11.4.7	针对端侧并发连接数的流控 .....	187
11.5	JVM 相关性能优化 .....	189
11.5.1	GC 调优 .....	189
11.5.2	其他优化手段 .....	193
11.6	总结 .....	193
<b>第 12 章</b>	<b>静态检查修改不当引起性能下降案例 .....</b>	<b>195</b>
12.1	Edge Service 性能严重下降问题 .....	195
12.1.1	Edge Service 热点代码分析 .....	195



12.1.2	静态检查问题不是简单的一改了之 .....	197
12.1.3	问题反思和改进 .....	200
12.2	克隆和浅拷贝 .....	201
12.2.1	浅拷贝存在的问题 .....	201
12.2.2	Netty 的对象拷贝实现策略 .....	203
12.3	总结 .....	204
第 13 章	Netty 性能统计误区案例 .....	205
13.1	时延毛刺排查相关问题 .....	205
13.1.1	时延毛刺问题初步分析 .....	205
13.1.2	服务调用链改进 .....	207
13.1.3	都是同步思维惹的祸 .....	208
13.1.4	正确的消息发送速度性能统计策略 .....	209
13.1.5	常见的消息发送性能统计误区 .....	212
13.2	Netty 关键性能指标采集策略 .....	212
13.2.1	Netty I/O 线程池性能指标 .....	213
13.2.2	Netty 发送队列积压消息数 .....	214
13.2.3	Netty 消息读取速度性能统计 .....	215
13.3	总结 .....	215
第 14 章	gRPC 的 Netty HTTP/2 实践案例 .....	216
14.1	gRPC 基础入门 .....	216
14.1.1	RPC 框架简介 .....	216
14.1.2	当前主流的 RPC 框架 .....	218
14.1.3	gRPC 框架特点 .....	218
14.1.4	为什么选择 HTTP/2 .....	219
14.2	gRPC Netty HTTP/2 服务端工作机制 .....	220
14.2.1	Netty HTTP/2 服务端创建原理和源码分析 .....	220
14.2.2	服务端接收 HTTP/2 请求消息原理和源码分析 .....	224
14.2.3	服务端发送 HTTP/2 响应消息原理和源码分析 .....	231
14.3	gRPC Netty HTTP/2 客户端工作机制 .....	234
14.3.1	Netty HTTP/2 客户端创建原理和源码分析 .....	235
14.3.2	客户端发送 HTTP/2 请求消息原理和源码分析 .....	238

14.3.3	客户端接收 HTTP/2 响应消息原理和源码分析 .....	242
14.4	gRPC 消息序列化机制 .....	243
14.4.1	Google Protobuf 简介 .....	243
14.4.2	消息的序列化原理和源码分析 .....	244
14.4.3	消息的反序列化原理和源码分析 .....	245
14.5	gRPC 线程模型 .....	246
14.5.1	服务端线程模型 .....	246
14.5.2	客户端线程模型 .....	247
14.5.3	线程模型总结 .....	248
14.6	总结 .....	249
第 15 章	Netty 事件触发策略使用不当案例 .....	250
15.1	channelReadComplete 方法被调用多次问题 .....	250
15.1.1	ChannelHandler 调用问题 .....	250
15.1.2	生产环境问题模拟重现 .....	252
15.2	ChannelHandler 使用的一些误区总结 .....	255
15.2.1	channelReadComplete 方法调用 .....	255
15.2.2	ChannelHandler 职责链调用 .....	257
15.3	总结 .....	258
第 16 章	Netty 流量整形应用案例 .....	259
16.1	Netty 流量整形功能 .....	259
16.1.1	通用的流量整形功能简介 .....	260
16.1.2	Netty 流量整形功能简介 .....	260
16.2	Netty 流量整形应用 .....	261
16.2.1	流量整形示例代码 .....	261
16.2.2	流量整形功能测试 .....	263
16.3	Netty 流量整形工作机制 .....	264
16.3.1	流量整形工作原理和源码分析 .....	264
16.3.2	并发编程在流量整形中的应用 .....	271
16.3.3	使用流量整形的一些注意事项总结 .....	274
16.4	总结 .....	278

第 17 章	Netty SSL 应用案例 .....	279
17.1	Netty SSL 功能简介 .....	279
17.1.1	SSL 安全特性 .....	280
17.1.2	Netty SSL 实现机制 .....	281
17.2	Netty 客户端 SSL 握手超时问题 .....	282
17.2.1	握手超时原因定位 .....	282
17.2.2	Netty SSL 握手问题定位技巧 .....	283
17.3	SSL 握手性能问题 .....	284
17.3.1	SSL 握手性能热点分析 .....	284
17.3.2	缓存和对象池 .....	285
17.4	SSL 事件监听机制 .....	286
17.4.1	握手成功事件 .....	286
17.4.2	SSL 连接关闭事件 .....	286
17.5	总结 .....	287
第 18 章	Netty HTTPS 服务端高并发宕机案例 .....	288
18.1	Netty HTTPS 服务端宕机问题 .....	288
18.1.1	客户端大量超时 .....	288
18.1.2	服务端内存泄漏原因分析 .....	289
18.1.3	NioSocketChannel 泄漏原因探究 .....	290
18.1.4	高并发场景下缺失的可靠性保护 .....	292
18.2	功能层面的可靠性优化 .....	294
18.2.1	Netty HTTPS 服务端可靠性优化 .....	295
18.2.2	HTTPS 客户端优化 .....	296
18.3	架构层面的可靠性优化 .....	297
18.3.1	端到端架构问题剖析 .....	297
18.3.2	HTTP Client 切换到 NIO .....	298
18.3.3	同步 RPC 调用切换到异步调用 .....	299
18.3.4	协议升级到 HTTP/2 .....	303
18.4	总结 .....	307

第 19 章	MQTT 服务接入超时案例	308
19.1	MQTT 服务接入超时问题	308
19.1.1	生产环境问题现象	308
19.1.2	连接数膨胀原因分析	309
19.1.3	无效连接的关闭策略	309
19.1.4	问题总结	310
19.2	基于 Netty 的可靠性设计	311
19.2.1	业务定制 I/O 异常	311
19.2.2	链路的有效性检测	312
19.2.3	内存保护	313
19.3	总结	315
第 20 章	Netty 实践总结	316
20.1	Netty 学习策略	316
20.1.1	入门知识准备	316
20.1.2	Netty 入门学习	319
20.1.3	项目实践	319
20.1.4	Netty 源码阅读策略	319
20.2	Netty 故障定位技巧	320
20.2.1	接收不到消息	320
20.2.2	内存泄漏	321
20.2.3	性能问题	322
20.3	总结	322

## 第 1 章

---

# Netty 服务端意外退出案例

在使用 Netty 进行服务端程序开发时，主要涉及端口监听、EventLoop 线程池创建、NioServerSocketChannel 和 ChannelPipeline 初始化等。初学者如果对服务端相关类库的工作原理和用法不熟悉，则会导致程序启动或者退出发生问题。本章从一个初学者容易犯错的案例展开分析，抽丝剥茧，让大家更好地掌握 Netty 服务端启停的原理和技巧，避免在工作中犯类似的错误。

## 1.1 Netty 服务端意外退出问题

---

**案例 1** 通过阻塞方式绑定监听端口，启动服务端之后，没发生任何异常，程序退出，代码示例如下：

---

```
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup, workerGroup)
  .channel(NioServerSocketChannel.class)
  .option(ChannelOption.SO_BACKLOG, 100)
  .handler(new LoggingHandler(LogLevel.INFO))
```



```
        .childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception {
                ChannelPipeline p = ch.pipeline();
                p.addLast(new LoggingHandler(LogLevel.INFO));
            }
        });
    b.bind(18080).sync(); //用同步方式绑定服务端监听端口
} finally {
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
```

---

**案例 2** 对案例 1 进行排查时，发现没有监听 Close Future，于是对代码进行修改，还是会发生服务器套接字直接关闭、进程退出的问题，代码示例如下：

```
ChannelFuture f = b.bind(18080).sync();
f.channel().closeFuture().addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        //业务逻辑处理代码，此处省略
        logger.info(future.channel().toString() + " 链路关闭");
    }
});
} finally {
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
```

---

### 1.1.1 Java Daemon 线程简介

在分析上面两个案例之前，需要弄清楚 Java 进程退出的原理，首先了解下 Java 的 Daemon 线程。所谓守护线程（Daemon）就是运行在程序后台的线程，通常守护线程是由 JVM 创建的，用于辅助用户线程或者 JVM 工作，比较典型的如 GC 线程。用户创建的线

程也可以设置成 Daemon 线程（通常需要谨慎设置），程序的主线程（main 线程）不是守护线程。Daemon 线程在 Java 里面的定义是，如果虚拟机中只有 Daemon 线程运行，则虚拟机退出。

（1）虚拟机中可能同时有多个线程运行，只有当所有的非守护线程（通常都是用户线程）都结束的时候，虚拟机的进程才会结束，不管当前运行的线程是不是 main 线程。

（2）main 线程运行结束，如果此时运行的其他线程全部是 Daemon 线程，JVM 会使这些线程停止，同时退出。但是如果此时正在运行的其他线程有非守护线程，那么必须等所有的非守护线程结束，JVM 才会退出。

看一下 Daemon 线程工作示例，主线程执行完，只有 Daemon 线程，进程退出，代码如下：

---

```
public static void main(String[] args)
    throws IllegalArgumentException, InterruptedException {
    long startTime = System.nanoTime();
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                TimeUnit.DAYS.sleep(Long.MAX_VALUE);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }, "Daemon-T");
    t.setDaemon(true);
    t.start();
    TimeUnit.SECONDS.sleep(15);
    System.out.println("系统退出，程序执行" + (System.nanoTime() - startTime) /
        1000 / 1000 + " s");
}
```

---

程序运行 15s 之后，进程正常退出，如图 1-1 所示。

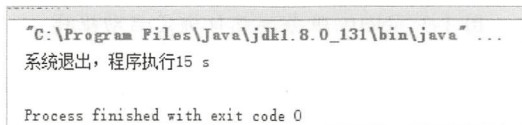


图 1-1 Daemon 线程退出执行结果

启动非 Daemon 线程，即使 main 线程执行完成，进程也不会退出，代码示例如下：

```
{
    //代码省略，同 DaemonT1
    t.setDaemon(false); //也可以不设置 Daemon 属性，默认为 false
    //代码省略
}
```

程序执行结果如图 1-2 所示，尽管 main 线程已经执行完成，但是 JVM 进程并没有退出。

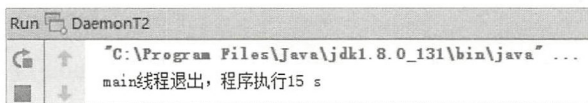


图 1-2 非 Daemon 线程退出执行结果

### 1.1.2 Netty 服务端启动原理

在 Netty 中，通过 `bootstrap.bind(PORT).sync().channel()` 方法绑定服务端端口，并不是在调用方的线程（示例为 main 线程）中执行，而是通过 `NioEventLoop` 线程执行，它的启动堆栈如图 1-3 所示。

```
DefaultChannelPipeline$HeadHandler.bind(ChannelHandlerContext, SocketAddress, ChannelPromi
ChannelHandlerInvokerUtil.invokeBindNow(ChannelHandlerContext, SocketAddress, ChannelPro
DefaultChannelHandlerInvoker.invokeBind(ChannelHandlerContext, SocketAddress, ChannelPro
DefaultChannelHandlerContext.bind(SocketAddress, ChannelPromise) line: 366
LoggingHandler.bind(ChannelHandlerContext, SocketAddress, ChannelPromise) line: 249
ChannelHandlerInvokerUtil.invokeBindNow(ChannelHandlerContext, SocketAddress, ChannelPro
DefaultChannelHandlerInvoker.invokeBind(ChannelHandlerContext, SocketAddress, ChannelPro
DefaultChannelHandlerContext.bind(SocketAddress, ChannelPromise) line: 366
DefaultChannelPipeline.bind(SocketAddress, ChannelPromise) line: 898
NioServerSocketChannel(AbstractChannel).bind(SocketAddress, ChannelPromise) line: 189
AbstractBootstrap$2.run() line: 309
NioEventLoop(SingleThreadEventExecutor).runAllTasks(long) line: 319
NioEventLoop.run() line: 355
```

图 1-3 Netty 服务端端口绑定启动堆栈

最终的执行结果其实就是调用了 Java NIO Socket 的端口绑定操作：

```
javaChannel().socket().bind(localAddress, config.getBacklog());
```

端口绑定操作执行完成之后，main 函数就不会阻塞，如果后续没有同步代码，main 线程就会退出。main 线程退出是否意味着 JVM 进程一定退出？并非如此，只有所有非守护线程全部执行完成，进程才会退出。此时系统中还存在其他非守护线程在运行吗？通过线程堆栈可以直观地查询各线程的运行状态，如图 1-4 所示。

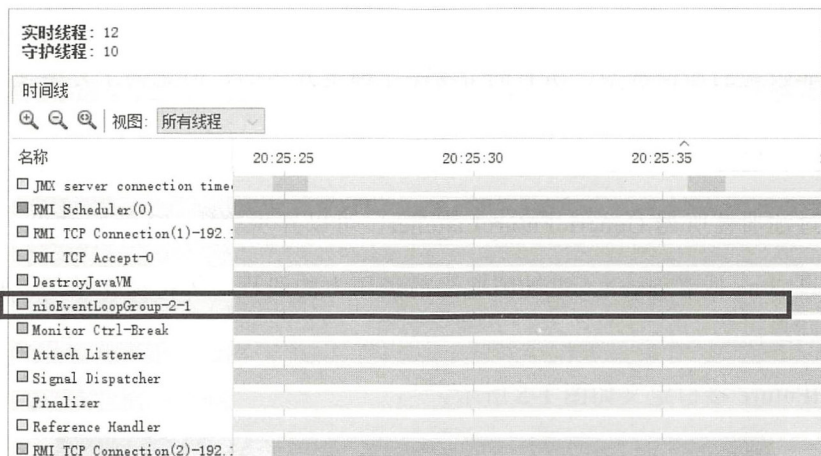


图 1-4 Netty 服务端线程堆栈

main 线程已经运行结束，但是 Netty 的 NioEventLoop 还处于运行状态，因此 JVM 进程并没有退出。

通过对 NioEventLoop 源码进行分析，可以明确如下几点。

- (1) NioEventLoop 是非守护线程。
- (2) NioEventLoop 运行之后，不会主动退出。
- (3) 只有调用 shutdown 系列方法，NioEventLoop 才会退出。

按照上面的分析，即使 main 函数执行结束，Netty 服务端启动之后进程也不应该退出，但为什么又退出了呢？仔细查看案例的代码，发现退出的原因有两点。

(1) 调用 `b.bind(18080).sync()` 之后，尽管它会同步阻塞，等待端口绑定结果，但是端口绑定执行得非常快，完成后程序就继续向下执行。

(2) 程序在 finally 里面执行了 `bossGroup.shutdownGracefully()` 和 `workerGroup.shutdownGracefully()`，它同时会关闭服务端的 TCP 连接接入线程池（bossGroup）和处理客户端网

络 I/O 读写的工作线程池（workerGroup），关闭之后，NioEventLoop 线程退出，整个系统的非守护线程就全部执行完成了，此时 main 函数主线程也早已执行完，因此 JVM 就会退出。因为调用的是 Netty 的优雅退出接口（shutdownGracefully），所以整个退出过程并没有发生异常。

出现案例 2 的问题，主要原因是使用者并没有掌握 Netty 的 ChannelFuture 机制，Netty 是一个异步非阻塞的通信框架，所有的 I/O 操作都是异步的，但是为了方便使用，例如在有些场景下应用需要同步阻塞等待一些 I/O 操作的结果，所以提供了 ChannelFuture，它主要提供以下两种能力。

（1）通过注册监听器 GenericFutureListener，可以异步等待 I/O 执行结果。

（2）通过 sync 或者 await，主动阻塞当前调用方的线程，等待操作结果，也就是通常说的异步转同步。

ChannelFuture 接口定义如图 1-5 所示。

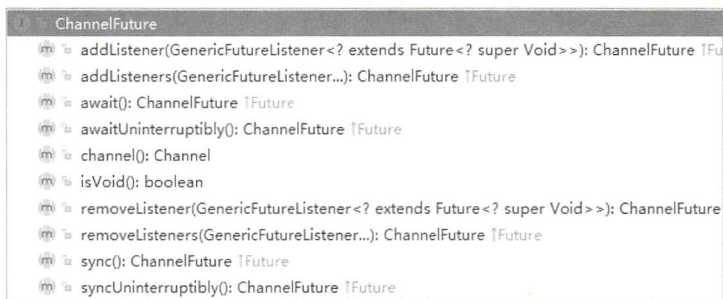


图 1-5 ChannelFuture 接口定义

案例 2 中增加了服务端连接关闭的监听事件之后，不会阻塞 main() 线程的执行，端口绑定成功之后，main 线程继续向下执行，由于在 finally 中增加了线程池关闭代码，NioEventLoop 线程主动退出，系统中没有正在运行的非守护线程了，所以 JVM 进程退出。

### 1.1.3 如何防止 Netty 服务端意外退出

分析清楚原因之后，我们可以通过不同的修改策略来防止 Netty 服务端意外退出。

（1）程序监听 NioServerSocketChannel 的关闭事件并同步阻塞 main 函数，我们对服务





端代码进行一些改造，代码示例如下：

```
{  
    //代码省略，同 DaemonT1  
    ChannelFuture f = b.bind(18080).sync();  
    f.channel().closeFuture().sync();  
    //代码省略，同 DaemonT1  
}
```

程序执行结果如图 1-6 所示，可以看出 main 函数处于阻塞状态，这样后续的 shutdownGracefully 方法就不会被执行，程序也不再退出。

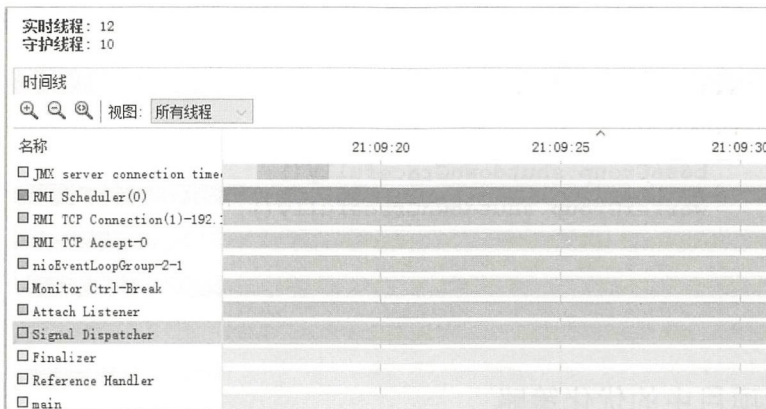


图 1-6 阻塞在 CloseFuture 中的线程状态

打印线程堆栈，发现 main 函数被阻塞在 CloseFuture 中，等待 Channel 关闭。

```
"main" #1 prio=5 os_prio=0 tid=0x00000000028de000 nid=0x854 in Object.wait() [0x0000000002dee000]  
  java.lang.Thread.State: WAITING (on object monitor)  
    at java.lang.Object.wait(Native Method)  
    - waiting on <0x00000000ecc26bb0> (a io.netty.channel.AbstractChannel$CloseFuture)  
    at java.lang.Object.wait(Object.java:502)  
    at io.netty.util.concurrent.DefaultPromise.await(DefaultPromise.java:231)  
    - locked <0x00000000ecc26bb0> (a io.netty.channel.AbstractChannel$CloseFuture)  
    at io.netty.channel.DefaultChannelPromise.await(DefaultChannelPromise.java:131)  
    at io.netty.channel.DefaultChannelPromise.await(DefaultChannelPromise.java:30)  
    at io.netty.util.concurrent.DefaultPromise.sync(DefaultPromise.java:337)  
    at io.netty.channel.DefaultChannelPromise.sync(DefaultChannelPromise.java:119)  
    at io.netty.channel.DefaultChannelPromise.sync(DefaultChannelPromise.java:30)  
    at io.netty.cases.chapter.demos.EchoExitServer3.main(EchoExitServer3.java:36)
```

图 1-7 阻塞在 CloseFuture 中的线程堆栈



(2) 注释掉 `bossGroup.shutdownGracefully()`和 `workerGroup.shutdownGracefully()`，改为在链路关闭时再释放线程池和连接句柄，代码示例如下：

---

```
//代码省略，同 DaemonT1
ChannelFuture f = b.bind(18080).sync();
    f.channel().closeFuture().addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws
Exception {
            //业务逻辑处理代码，此处省略
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
            logger.info(future.channel().toString() + " 链路关闭");
        }
    });
    } finally {
//        bossGroup.shutdownGracefully();
//        workerGroup.shutdownGracefully();
    }
}
```

---

### 1.1.4 实际项目中的优化策略

初学者很容易出现上述案例中的错误用法，但在实际项目中，很少通过 `main` 函数直接调用 Netty 服务端，业务往往是通过某种容器（例如 Tomcat、SpringBoot 等）拉起进程，然后通过容器启动来初始化各种业务资源。因此，不需要担心 Netty 服务端意外退出，启动 Netty 服务端比较容易犯的错误是采用同步的方式调用 Netty，导致初始化 Netty 服务端的业务线程被阻塞，举例如下。

错误用法：这种用法会导致调用方的线程一直被阻塞，直到服务端监听句柄关闭。

- ◎ 初始化 Netty 服务端。
- ◎ 同步阻塞等待服务端端口关闭。
- ◎ 释放 I/O 线程资源和句柄等。



◎ 调用方线程被释放。

正确用法：服务端启动之后注册监听器监听服务端句柄关闭事件，待服务端关闭之后异步调用 `shutdownGracefull` 释放资源，这样调用方线程就可以快速返回，不会被阻塞。

◎ 初始化 Netty 服务端。

◎ 绑定监听端口。

◎ 向 `CloseFuture` 注册监听器，在监听器中释放资源。

◎ 调用方线程返回。

很多开发者习惯了写同步代码，在使用 Netty 之后仍然采用同步阻塞的方式来调用 Netty，尽管功能上也可以正常使用，但是违背了 Netty 的异步设计理念，线程执行效率并不高。

当系统退出时，建议通过调用 `EventLoopGroup` 的 `shutdownGracefully` 来完成内存队列中积压消息的处理、链路的关闭和 `EventLoop` 线程的退出，以实现停机不中断业务（备注：单靠 Netty 框架实际上无法 100% 保证，需要应用配合来实现）。

## 1.2 Netty 优雅退出机制

在 Linux 上通常会通过 `kill -9 pid` 的方式强制将某个进程杀掉，这种方式简单高效，因此很多程序的停止脚本经常会使用 `kill -9 pid` 的方式。

无论是 Linux 的 `kill -9 pid` 还是 Windows 的 `taskkill /f /pid` 强制进程退出，都会带来一些副作用，对应用软件而言其效果等同于突然掉电，可能会导致如下问题。

- （1）缓存中的数据尚未持久化到磁盘中，导致数据丢失。
- （2）正在进行文件的写（write）操作，没有更新完成，突然退出，导致文件损坏。
- （3）线程的消息队列中尚有接收到的请求消息还没来得及处理，导致请求消息丢失。
- （4）数据库操作已经完成，例如账户余额更新，准备返回应答消息给客户端时，消息尚在通信线程的发送队列中排队等待发送，进程强制退出导致应答消息没有返回给客户端，



客户端发起超时重试，会带来重复更新问题。

(5) 句柄资源没有及时释放等其他问题。

### 1.2.1 Java 优雅退出机制

Java 的优雅停机通常通过注册 JDK 的 ShutdownHook 来实现，当系统接收到退出指令时，首先标记系统处于退出状态，不再接收新的消息，然后将积压的消息处理完，最后调用资源回收接口将资源销毁，各线程退出执行。

通过 JDK ShutdownHook 实现的优雅退出代码示例如下：

```
Runtime.getRuntime().addShutdownHook(new java.lang.Thread()->
{
    System.out.println("ShutdownHook execute start...");
    System.out.println("Netty NioEventLoopGroup shutdownGracefully...");
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("ShutdownHook execute end...");
}, "");
TimeUnit.SECONDS.sleep(7);
System.exit(0);
}
```

它的执行结果如图 1-8 所示。

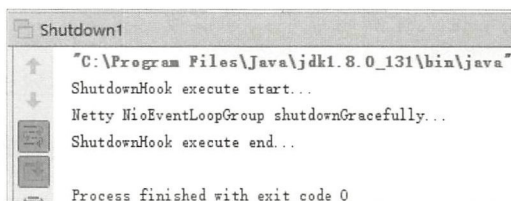


图 1-8 ShutdownHook 执行结果

除了注册 ShutdownHook，还可以通过监听信号量并注册 SignalHandler 的方式实现优雅退出，它的工作原理如图 1-9 所示。

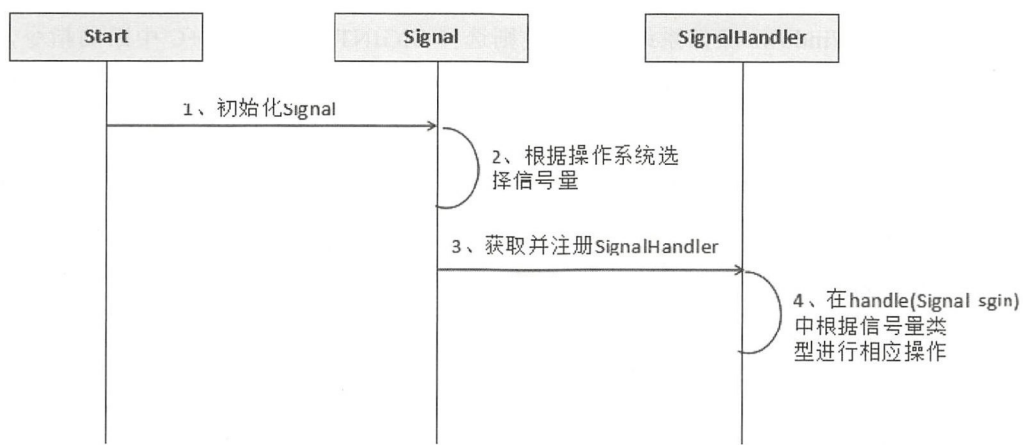


图 1-9 SignalHandler 的工作原理

(1) 启动应用进程的时候，初始化 Signal 实例，代码如下：

```
Signal sig = new Signal(getOSSignalType());
```

其中 Signal 构造函数的参数为 String 字符串，它代表了操作系统支持的信号量列表(此处注意：不同操作系统支持的信号量不同)，如表 1-1 所示为 Linux 支持的一些常用信号量。

表 1-1 Linux 支持的一些常用信号量

信 号 名 称	用 途
SIGKILL	终止进程，强制杀死进程
SIGTERM	终止进程，软件终止信号
SIGTSTP	停止进程，终端来的停止信号
SIGUSR1	终止进程，用户定义信号 1
SIGUSR2	终止进程，用户定义信号 2
SIGINT	终止进程，中断进程
SIGQUIT	建立 core 文件终止进程，并且生成 core 文件

(2) 根据操作系统的名称来获取对应的信号名称：





---

```
System.getProperties().getProperty("os.name").  
    toLowerCase().startsWith("win") ? "INT" : "TERM"
```

---

判断是否是 Windows 操作系统，如果是则选择 SIGINT，接收 Ctrl+C 中断的指令，否则选择 TERM 信号，接收 SIGTERM（等价于 kill pid）指令（备注：这里仅是支持 Windows 和 Linux 操作系统的代码示例）。

（3）将实例化之后的 SignalHandler 注册到 JDK 的 Signal，一旦 Java 进程接收到 kill pid 或 Ctrl+C，则回调 handle 接口：

---

```
Signal.handle(sig, shutdownHandler);
```

---

（4）在接收到信号回调的 handle 接口中，判断信号量的类型，如果是 SIGTERM，则执行应用的优雅退出操作，对于 Netty，需要调用 EventLoopGroup 的 shutdownGracefully 方法，释放通信层资源。

## 1.2.2 Java 优雅退出的注意点

对于通过注册 ShutdownHook 实现的优雅退出，需要注意如下几点，防止踩坑。

（1）ShutdownHook 在某些情况下并不会被执行，例如 JVM 崩溃、无法接收信号量和 kill -9 pid 等。

（2）当存在多个 ShutdownHook 时，JVM 无法保证它们的执行先后顺序。

（3）在 JVM 关闭期间不能动态添加或者去除 ShutdownHook。

（4）不能在 ShutdownHook 中调用 System.exit()，它会卡住 JVM，导致进程无法退出。

对于采用注册 SignalHandler 实现优雅退出的程序，在 handle 接口中一定要避免阻塞操作，否则它会导致已经注册的 ShutdownHook 无法执行，系统也无法退出，代码示例如下：

---

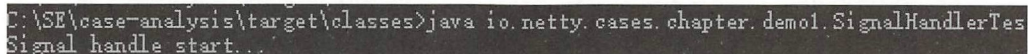
```
Signal sig = new Signal("INT");//以 Windows 操作系统为例  
Signal.handle(sig, (s)->{  
    System.out.println("Signal handle start...");
```

---



```
try {
    TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);
} catch (InterruptedException e) {
    e.printStackTrace();
}
});
Runtime.getRuntime().addShutdownHook(new Thread(() ->
{
    System.out.println("ShutdownHook execute start...");
    System.out.println("Netty NioEventLoopGroup shutdownGracefully...");
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("ShutdownHook execute end...");
}, ""));
}
```

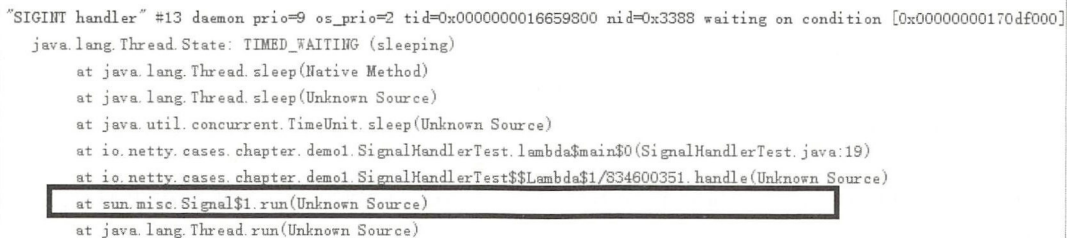
在 Windows 上按 Ctrl+C 组合键停止进程，执行结果如图 1-10 所示。



```
C:\SE\case-analysis\target\classes>java io.netty.cases.chapter.demo1.SignalHandlerTest
Signal handle start...
```

图 1-10 模拟 SignalHandler 阻塞执行结果

通过线程堆栈分析，发现代码阻塞在 SIGINT handler 中，如图 1-11 所示。



```
"SIGINT handler" #13 daemon prio=9 os_prio=2 tid=0x0000000016659800 nid=0x3388 waiting on condition [0x00000000170df000]
java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Unknown Source)
    at java.util.concurrent.TimeUnit.sleep(Unknown Source)
    at io.netty.cases.chapter.demo1.SignalHandlerTest.lambda$main$0(SignalHandlerTest.java:19)
    at io.netty.cases.chapter.demo1.SignalHandlerTest$$Lambda$1/834600351.handle(Unknown Source)
    at sun.misc.Signal$1.run(Unknown Source)
    at java.lang.Thread.run(Unknown Source)
```

图 1-11 模拟 SignalHandler 阻塞线程堆栈



由于 SignalHandler 发生了阻塞, 导致 ShutdownHook 无法执行, 因此没有打印 ShutdownHook 执行相关日志。如果 SignalHandler 执行的操作比较耗时, 建议异步或放到 ShutdownHook 中执行。

### 1.2.3 Netty 优雅退出机制

在实际项目中, Netty 作为高性能的异步 NIO 通信框架, 往往作为基础通信框架负责各种协议的接入、解析和调度等, 例如在 RPC 和分布式服务框架中, 往往会使用 Netty 作为内部私有协议的基础通信框架。

当应用进程优雅退出时, 作为通信框架的 Netty 也需要优雅退出, 主要原因如下。

- (1) 尽快释放 NIO 线程和句柄等资源。
- (2) 如果使用 flush 做批量消息发送, 需要将积压在发送队列中的待发送消息发送完成。
- (3) 正在写或者读的消息, 需要继续处理。
- (4) 设置在 NioEventLoop 线程调度器中的定时任务, 需要执行或清理。

下面看下 Netty 优雅退出涉及的主要操作和资源对象, 如图 1-12 所示。

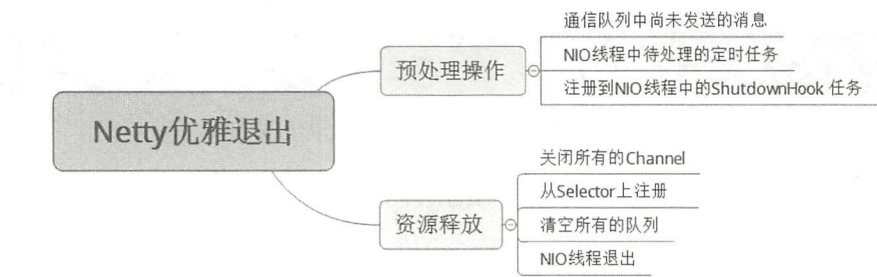


图 1-12 Netty 优雅退出涉及的主要操作和资源对象

Netty 优雅退出总结起来有如下三大类操作。

- (1) 把 NIO 线程的状态位设置成 ST\_SHUTTING\_DOWN, 不再处理新的消息 (不允许再对外发送消息)。
- (2) 退出前的预处理操作: 把发送队列中尚未发送或者正在发送的消息发送完 (备注:



不保证能够发送完)、把已经到期或在退出超时之前到期的定时任务执行完成、把用户注册到 NIO 线程的退出 Hook 任务执行完成。

(3) 资源的释放操作: 所有 Channel 的释放、多路复用器的去注册和关闭、所有队列和定时任务的清空取消, 最后是 EventLoop 线程的退出。

Netty 优雅退出的接口和总入口是 EventLoopGroup, 调用它的 shutdownGracefully 方法即可, 相关代码示例如下:

```
bossGroup.shutdownGracefully();  
workerGroup.shutdownGracefully();
```

除了无参的 shutdownGracefully 方法, 还可以指定退出的超时时间和周期, 相关接口定义如图 1-13 所示。

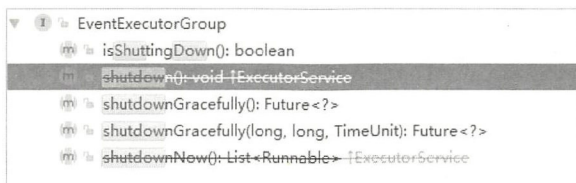


图 1-13 EventLoopGroup 优雅退出相关接口定义

其中, 强制退出已经被标注为废弃, 在实际项目中尽量不要使用。当 JVM 的 ShutdownHook 被触发之后, 调用所有 EventLoopGroup 实例的 shutdownGracefully 方法进行优雅退出。由于 Netty 自身对优雅退出有较完善的支持, 所以实现起来相对比较简单。

## 1.2.4 Netty 优雅退出原理和源码分析

Netty 优雅退出涉及线程组、NIO 线程、Channel 和定时任务等, 底层实现细节比较复杂, 下面我们就层层分解, 通过源码分析来了解它的实现原理。

### 1. NioEventLoopGroup

NioEventLoopGroup 实际上是 NioEventLoop 线程组, 它的优雅退出比较简单, 可直接遍历 EventLoop 数组, 循环调用它们的 shutdownGracefully 方法, 源码如下 (MultithreadEventExecutorGroup 的 shutdownGracefully 方法):



---

```
for (EventExecutor l: children) {  
    l.shutdownGracefully(quietPeriod, timeout, unit);  
}
```

---

## 2. NioEventLoop

调用 NioEventLoop 的 shutdownGracefully 方法,首先要修改线程状态为正在关闭状态,它的实现在父类 SingleThreadEventExecutor 中,需要注意的是,修改线程状态位时要对并发调用做保护,因为调用 shutdownGracefully 方法可能由 NioEventLoop 线程发起,也可能多个应用线程并发执行。对于线程状态的修改需要做并发保护,最简单的策略就是加锁,或者采用原子类加自旋的方式避免加锁,Netty 5 采用的是加锁策略,Netty 4 则采用后者,Netty 4 的处理逻辑如下:

---

```
for (;;) {  
    if (isShuttingDown()) {  
        return terminationFuture();  
    }  
    //代码省略  
    if (STATE_UPDATER.compareAndSet(this, oldState, newState)) {  
        break;  
    }  
}
```

---

从上述代码可以看出,采用 AtomicIntegerFieldUpdater 的 compareAndSet 对新老线程状态进行修改,如果在修改当前线程时发现状态已经被别的线程修改过,则继续自旋,直到发现线程状态已经处于 ST\_SHUTTING\_DOWN、ST\_SHUTDOWN 和 ST\_TERMINATED 状态,或者自己的更新操作成功,才会退出循环。

完成状态修改之后,剩下的操作主要在 NioEventLoop 中进行,代码示例如下:

---

```
if (isShuttingDown()) {  
    closeAll();  
    if (confirmShutdown()) {  
        return;  
    }  
}
```

---



继续分析 `closeAll` 的实现，它的原理是把注册在 `selector` 上的所有 `Channel` 都关闭，核心代码示例如下：

---

```
for (SelectionKey k: keys) {
    Object a = k.attachment();
    if (a instanceof AbstractNioChannel) {
        channels.add((AbstractNioChannel) a);
    } else {
        k.cancel();
        NioTask<SelectableChannel>task=(NioTask<SelectableChannel>) a;
        invokeChannelUnregistered(task, k, null);
    }
}

for (AbstractNioChannel ch: channels) {
    ch.unsafe().close(ch.unsafe().voidPromise());
}
```

---

循环调用 `Channel Unsafe` 的 `close` 方法，下面跳转到 `Unsafe` 中，对 `close` 方法进行分析。

### 3. AbstractUnsafe

`AbstractUnsafe` 的 `close` 方法主要完成如下几个功能。

(1) 判断当前链路是否有消息正在发送，如果有则将 `SelectionKey` 的去注册操作封装成 `Task` 放到 `eventLoop` 中稍后再执行：

---

```
if (inFlush0) {
    invokeLater(new Runnable() {
        @Override
        public void run() {
            fireChannelInactiveAndDeregister(wasActive);
        }
    });
} else {
    fireChannelInactiveAndDeregister(wasActive);
}
```

---



(2) 将发送队列清空，不再允许发送新的消息：

---

```
final boolean wasActive = isActive();  
    final ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;  
    this.outboundBuffer = null;
```

---

(3) 调用 `NioSocketChannel` 的 `doClose` 方法，关闭链路：

---

```
super.doClose();  
javaChannel().close();
```

---

(4) 调用 `pipeline` 的 `fireChannelInactive`，触发链路关闭通知事件：

---

```
fireChannelInactiveAndDeregister(wasActive);
```

---

(5) 调用 `AbstractNioChannel` 的 `Deregister`，从多路复用器上取消 `selectionKey`：

---

```
protected void doDeregister() throws Exception {  
    eventLoop().cancel(selectionKey());  
}
```

---

(6) 调用 `ChannelOutboundBuffer` 的 `close` 方法，释放发送队列中所有尚未完成发送的 `ByteBuf`（关闭之前没有被 flushed 的 message），等待 GC：

---

```
if (!e.cancelled) {  
    ReferenceCountUtil.safeRelease(e.msg);  
    safeFail(e.promise, cause);  
}  
  
//代码省略  
clearNioBuffers();
```

---

执行完资源释放和连接关闭操作之后，`NioEventLoop` 还有扫尾工作需要执行，`NioEventLoop` 除了 I/O 读写，还负责定时任务执行、`ShutdownHook`（备注：此处非 JDK 原生的 `ShutdownHook`）的执行等，如果此时有到期的定时任务，即使 `Channel` 已经关闭，但是仍然需要继续执行，线程不能退出，下面继续分析 `TaskQueue` 的退出处理流程。

#### 4. TaskQueue

`NioEventLoop` 执行完 `closeAll()` 操作，需要调用 `confirmShutdown` 看是否真的可以退出，



它的判断逻辑如下（NioEventLoop run 方法）：

---

```

if (isShuttingDown()) {
    closeAll();
    if (confirmShutdown()) {
        return;
    }
}

```

---

在 confirmShutdown 方法中，执行如下操作。

（1）执行尚在 TaskQueue 中排队的 Task，代码示例如下：

---

```

do {
    fetchedAll = fetchFromScheduledTaskQueue();
    if (runAllTasksFrom(taskQueue)) {
        ranAtLeastOne = true;
    }
} while (!fetchedAll);

```

---

（2）执行注册到 NioEventLoop 中的 ShutdownHook，代码示例如下：

---

```

private boolean runShutdownHooks() {
    boolean ran = false;
    while (!shutdownHooks.isEmpty()) {
        List<Runnable> copy = new ArrayList<Runnable>(shutdownHooks);
        shutdownHooks.clear();
        for (Runnable task: copy) {
            try {
                task.run();
            } catch (Throwable t) {
                logger.warn("Shutdown hook raised an exception.", t);
            } finally {
                ran = true;
            }
        }
    }
}

```

---

（3）判断是否到达优雅退出的指定超时时间，如果达到或者过了超时时间，则立即退



出，代码示例如下：

---

```
if (isShutdown() || nanoTime - gracefulShutdownStartTime > gracefulShutdownTimeout) {  
    return true;  
}
```

---

(4) 如果没到达指定的超时时间，暂时不退出，每隔 100ms 检测一下是否有新的任务加入，有新任务则继续执行：

---

```
if (nanoTime - lastExecutionTime <= gracefulShutdownQuietPeriod) {  
    wakeup(true);  
    try {  
        Thread.sleep(100);  
    } catch (InterruptedException e) {  
        // 代码省略  
    }  
}
```

---

当 `confirmShutdown` 返回 `true`，`NioEventLoop` 线程正式退出，Netty 的优雅退出完成，代码示例如下（`NioEventLoop` 的 `run` 方法）：

---

```
for (;;) {  
    //代码省略  
    if (confirmShutdown()) {  
        return;  
    }  
    //代码省略  
}
```

---

## 1.2.5 Netty 优雅退出的一些误区

不同版本 Netty 优雅退出的实现策略不同，特别是大版本之间（Netty 3.X / 4.X / 5.X）的差异还是比较大的，但是都保证不了优雅退出时所有消息队列排队的消息能够处理完，主要原因如下。



(1) 待发送的消息：调用优雅退出方法之后，不会立即关闭链路。ChannelOutboundBuffer 中的消息可以继续发送，本轮发送操作执行完成之后，无论是否还有消息尚未发送出去，在下一轮的 Selector 轮询中，链路都将被关闭，没有发送完成的消息将会被释放和丢弃。

(2) 需要发送的新消息：由于应用线程可以随时通过调用 Channel 的 write 系列接口发送消息，即便 ShutdownHook 触发了 Netty 的优雅退出方法，在 Netty 优雅退出方法执行期间，应用线程仍然有可能继续调用 Channel 发送消息，这些消息将发送失败。

应用注册在 NioEventLoop 线程上的普通 Task、Scheduled Task（定时任务）和 ShutdownHook，也无法保证被完全执行，这取决于优雅退出超时时间和任务的数量，以及执行速度。

因此，应用程序的正确性不能完全依赖 Netty 的优雅退出机制，需要在应用层面做容错设计和处理。例如，服务端在返回响应之前关闭了，导致响应没有发送给客户端，这可能会触发客户端的 I/O 异常，或者恰好发生了超时异常，客户端需要对 I/O 或超时异常做容错处理，采用 Failover 重试其他可用的服务端，而不能寄希望于服务端永远正确。Netty 优雅退出更重要的是保证资源、句柄和线程的快速释放，以及相关对象的清理。

Netty 优雅退出通常用于应用进程退出时，在应用的 ShutdownHook 中调用 EventLoopGroup 的 shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit)接口，指定退出的超时时间，以防止因为一些任务执行被阻塞而无法退出。

## 1.3 总结

本章通过两个简单的案例分析，引出了信号量、Java Daemon 线程及 Netty 优雅退出相关知识。在实际项目中，知识往往是交叉在一起的，要想熟练掌握 Netty 服务端的启动和退出，编写更优雅和健壮的代码，需要重点掌握如下几个知识点：

- (1) 操作系统的信号量和 Java Deamon 线程工作机制。
- (2) Netty 的 NioEventLoop 线程工作原理。
- (3) Netty 优雅退出相关的几个核心类库。





## 第 2 章

---

# Netty 客户端连接池资源泄漏案例

随着硬件性能的不不断提升，多处理器和多网卡已经成为标配，为了充分利用硬件资源，应用程序通过并发编程、客户端和服务端创建多链路的方式提升性能。

通过连接池，客户端和服务端之间可以创建多个 TCP 连接，提升消息的收发能力，同时利用池化技术，可以重用连接，防止反复申请和释放连接，提高连接的使用率。在实际项目中，各类连接池被大量使用，例如数据库连接池、HTTP 连接池等。当使用 Netty 客户端创建连接池时，如果对 Netty 的客户端连接创建机制不熟悉，很可能导致线程膨胀、内存溢出等问题。

### 2.1 Netty 连接池资源泄漏问题

---

生产环境使用 Netty 作为客户端通信框架，为了提升性能，客户端与服务端之间创建了多条链路，同时客户端创建了一个 TCP 连接池，它会随着业务压力的变化动态调整连接池的大小。在测试环境中验证没有问题，到了生产环境中，在业务高峰期总会出现 OOM 问题，需要重启才能恢复。



### 2.1.1 连接池创建代码

为了便于分析说明，对实际代码中的连接池创建做简化处理，代码示例如下：

---

```
static void initClientPool(int poolSize) throws Exception
{
    for(int i = 0; i < poolSize; i++)
    {
        EventLoopGroup group = new NioEventLoopGroup();
        Bootstrap b = new Bootstrap();
        b.group(group)
            .channel(NioSocketChannel.class)
            .option(ChannelOption.TCP_NODELAY, true)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws
Exception {
                    ChannelPipeline p = ch.pipeline();
                    p.addLast(new LoggingHandler());
                }
            });
        ChannelFuture f = b.connect(HOST, PORT).sync();
        f.channel().closeFuture().addListener((r)->
        {
            group.shutdownGracefully();
        });
    }
}
```

---

备注：实际业务的连接池更复杂一些，还包含连接池的动态扩容和空闲连接的释放等。

### 2.1.2 内存溢出和线程膨胀

将连接池的连接数上限配置为 100，业务高峰期发生了 OOM 异常，业务需要重启才



能恢复，相关异常日志如图 2-1 所示。

```
Exception in thread "nioEventLoopGroup-77-1" Exception in thread "RMI TCP Connection(idle)" java.lang.OutOfMemoryError:
java.lang.OutOfMemoryError: GC overhead limit exceeded
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.net.InetAddress.<init>(InetAddress.java:308)
    at java.net.InetAddress.<init>(InetAddress.java:102)
    at sun.nio.ch.Net.localInetAddress(Native Method)
    at sun.nio.ch.Net.localAddress(Net.java:479)
    at sun.nio.ch.SocketChannelImpl.<init>(SocketChannelImpl.java:133)
    at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:266)
    at sun.nio.ch.PipeImpl$Initializer$LoopbackConnector.run(PipeImpl.java:135)
    at sun.nio.ch.PipeImpl$Initializer.run(PipeImpl.java:76)
    at sun.nio.ch.PipeImpl$Initializer.run(PipeImpl.java:61) <1 internal calls>
    at sun.nio.ch.PipeImpl.<init>(PipeImpl.java:171)
    at sun.nio.ch.SelectorProviderImpl.openPipe(SelectorProviderImpl.java:50)
```

图 2-1 连接池发生 OOM 异常日志

查看线程堆栈，发现存在大量的 EventLoopGroup 线程池，如图 2-2 所示。

筛选列	线程名称	NioEvent	<input checked="" type="checkbox"/> CPU 概要分析		
线程名称	线程状态	阻塞计数	CPU 总体占用率	死锁	
⌵ nioEventLoopGroup-46-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-45-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-44-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-43-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-42-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-41-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-40-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-39-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-38-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-37-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-36-1	RUNNABLE	0		否	
⌵ nioEventLoopGroup-35-1	RUNNABLE	0		否	

图 2-2 发现大量的 EventLoopGroup 线程池

从异常日志和线程资源占用看，导致内存泄漏的原因是应用创建了大量的 EventLoopGroup 线程池，每个 EventLoopGroup 线程池里面的 NioEventLoop 线程（对应线程名为 nioEventLoopGroup-XX-1）对应一个 TCP 连接，如果有 100 个连接则会创建 100 个线程，当系统压力大、连接数比较多时，就会导致 NioEventLoop 线程膨胀，因为每个线程本身需要占用一定的内存，再加上 NioEventLoop 线程及其成员变量、相关资源的内存占用，当系统堆内存不足时新连接创建就会失败，发生 OOM 异常。

通过代码分析发现，应用在初始化连接池时，采用每个客户端连接对应一个 EventLoopGroup 实例的方式，即每创建一个客户端连接，就会同时创建一个 NioEventLoop



线程来处理客户端连接及后续的网络读写操作，采用的策略是典型的一个 TCP 连接对应一个 NIO 线程的模式。当系统的连接数很多、堆内存又不足时，就会发生内存泄漏或者线程创建失败异常。错误的客户端线程模型如图 2-3 所示。

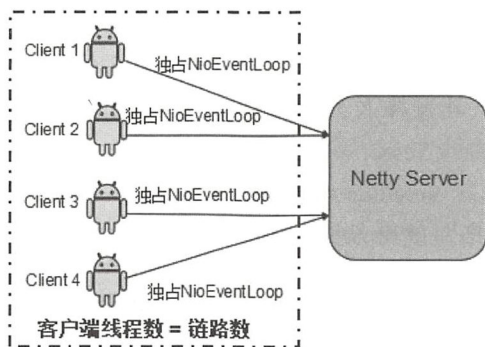


图 2-3 错误的客户端线程模型

### 2.1.3 错用 NIO 编程模式

前面连接池泄漏的原因是采用 BIO 模式来调用 NIO 通信框架，不仅没达到优化的效果，而且还发生了 OOM 异常。

在 BIO 模型中，由于网络的读写操作都是同步阻塞的，为了尽可能提升系统的吞吐量，往往采用一个线程对应一个连接的方式，它的服务端通信模型如图 2-4 所示。

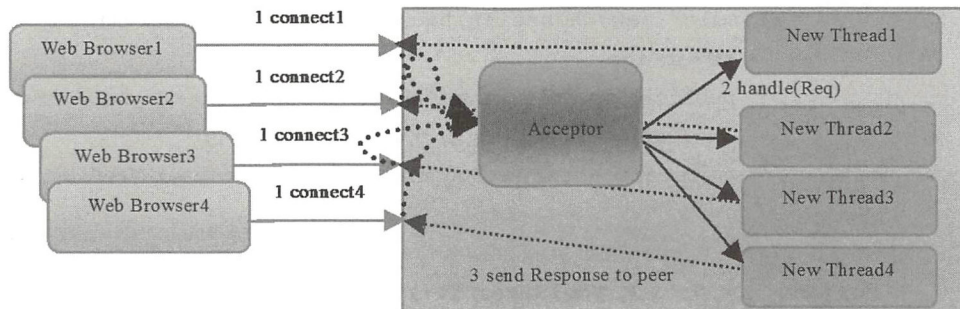


图 2-4 同步阻塞 I/O 服务端通信模型（一客户端一线程）

对于客户端，每个连接都会对应一个线程，对于服务端，则会为每个客户端连接分配

一个业务线程处理，形成 1:1 的对应关系。由于线程是 Java 虚拟机非常宝贵的系统资源，线程数膨胀之后，系统的性能将急剧下降，随着并发访问量的继续增大，系统会出现线程堆栈溢出、创建新线程失败等问题，并最终导致进程宕机或者僵死，不能对外提供服务。

作为高性能的 NIO 通信框架，Netty 解决了传统 BIO 模型遇到的问题。在 NIO 线程中聚合了多路复用器 Selector，Selector 会不断地轮询注册在其上的 Channel，如果某个 Channel 上面有新的 TCP 连接接入、读和写事件，这个 Channel 就处于就绪状态，会被 Selector 轮询出来，然后通过 SelectionKey 可以获取就绪 Channel 的集合，进行后续的 I/O 操作。由于一个多路复用器 Selector 可以同时轮询多个 Channel，这也就意味着只需要一个线程负责 Selector 的轮询，就可以接入成千上万的客户端。

在上节的案例代码中，错误地把 Netty 当作传统的 BIO 框架来使用，为每个连接都分配一个线程池，当连接数较多而堆内存不足时自然会发生 OOM 异常。

## 2.1.4 正确的连接池创建方式

创建多个连接时，EventLoopGroup 可以重用，优化之后的代码如下：

---

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap b = new Bootstrap();
b.group(group)
    .channel(NioSocketChannel.class)
    .option(ChannelOption.TCP_NODELAY, true)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) throws Exception {
            ChannelPipeline p = ch.pipeline();
            p.addLast(new LoggingHandler());
        }
    });
for(int i = 0; i < poolSize; i++)
{
    b.connect(HOST, PORT).sync();
}
```

---



修改之后，发现业务高峰期运行正常，没有再发生 OOM 问题，TCP 连接数如图 2-5 所示（Windows 示例）。

```
C:\Users\李林锋>netstat -an|find "ESTABLISHED"|find "18081" /c
200
```

图 2-5 TCP 连接数

查看系统线程资源占用，为 CPU 内核数的 2 倍，与 TCP 连接数无关，解决了线程膨胀问题，如图 2-6 所示。

筛选列	线程名称	线程状态	阻塞计数	CPU 总体占用率
	nio			
	nioEventLoopGroup-2-8	RUNNABLE	10	0.00%
	nioEventLoopGroup-2-7	RUNNABLE	9	0.00%
	nioEventLoopGroup-2-6	RUNNABLE	18	0.00%
	nioEventLoopGroup-2-5	RUNNABLE	13	0.00%
	nioEventLoopGroup-2-4	RUNNABLE	16	0.00%
	nioEventLoopGroup-2-3	RUNNABLE	12	0.00%
	nioEventLoopGroup-2-2	RUNNABLE	15	0.00%
	nioEventLoopGroup-2-1	RUNNABLE	15	0.00%

图 2-6 连接池 NIO 线程资源占用

修改之后正确的客户端线程模型如图 2-7 所示。

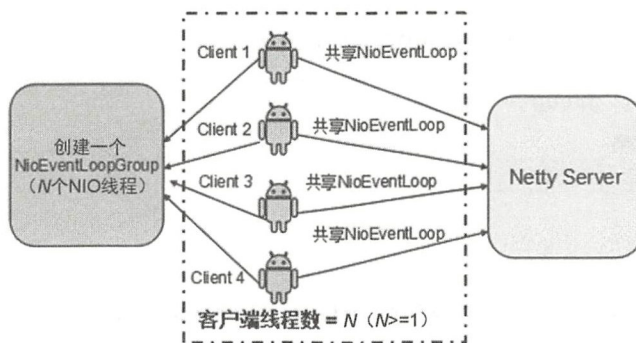


图 2-7 修改之后正确的客户端线程模型

尽管 Bootstrap 自身不是线程安全的，但是执行 Bootstrap 的连接操作是串行执行的，而且 `connect(String inetHost, int inetPort)` 方法本身是线程安全的，它会创建一个新的 `NioSocketChannel`，并从初始构造的 `EventLoopGroup` 中选择一个 `NioEventLoop` 线程执行真正的 Channel 连接操作，与执行 Bootstrap 的线程无关，所以通过一个 Bootstrap 连续发起多个连接操作是安全的，它的工作原理如图 2-8 所示。

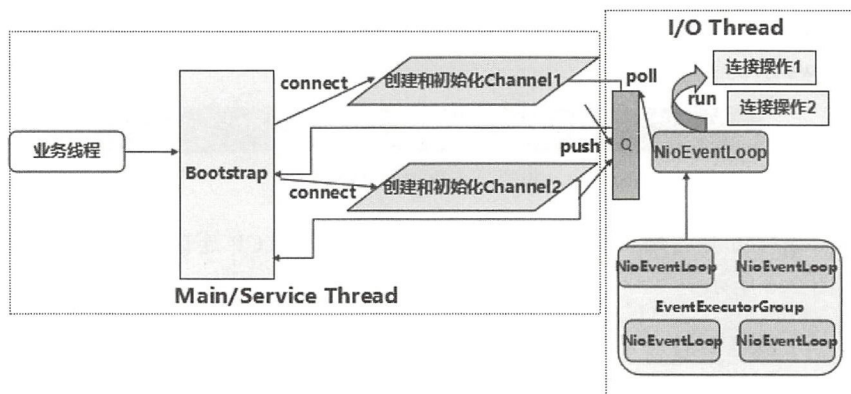


图 2-8 Bootstrap 工作原理

## 2.1.5 并发安全和资源释放

在同一个 Bootstrap 中连续创建多个客户端连接，需要注意的是，EventLoopGroup 是共享的，也就是说这些连接共用同一个 NIO 线程组 EventLoopGroup，当某个链路发生异常或者关闭时，只需要关闭并释放 Channel 本身即可，不能同时销毁 Channel 所使用的 NioEventLoop 和所在的线程组 EventLoopGroup，例如下面的代码就是错误的：

---

```
EventLoopGroup group = new NioEventLoopGroup();
//代码省略
for(int i = 0; i < poolSize; i++)
{
    ChannelFuture f = b.connect(HOST, PORT).sync();
    f.channel().closeFuture().addListener((r) ->
    {
        group.shutdownGracefully();
    });
}
```

---

需要指出的是，Bootstrap 不是线程安全的，因此在多个线程中并发操作 Bootstrap 是一件比较危险的事情，Bootstrap 是 I/O 操作工具类，它自身的逻辑处理非常简单，真正的 I/O 操作都是由 EventLoop 线程负责的，所以通常多线程操作同一个 Bootstrap 实例也是没

有意义的，而且容易出错，错误代码示例如下：

```
Bootstrap b = new Bootstrap();
//多线程并发执行初始化工作
```

## 2.2 Netty 客户端创建机制

只有更好地了解 Java NIO 客户端相关类库，以及 Netty Bootstrap 的功能和工作原理，才能在实际项目中更得心应手地使用 Netty。

### 2.2.1 Java NIO 客户端创建原理分析

原生 Java NIO 客户端创建流程如图 2-9 所示。

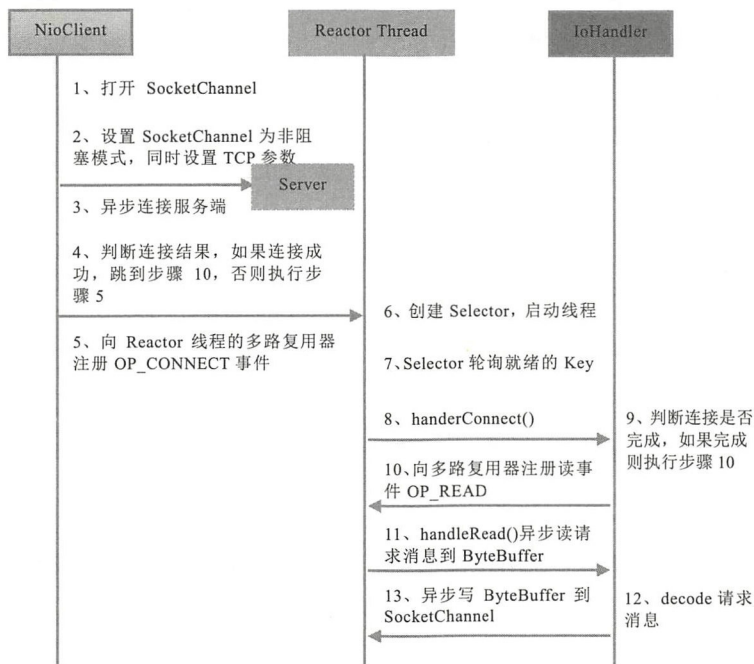


图 2-9 原生 Java NIO 客户端创建流程

主要流程介绍如下。

(1) 打开 `SocketChannel`，绑定客户端本地地址（可选，系统默认会随机分配一个可用的本地地址），示例代码如下：

---

```
SocketChannel clientChannel = SocketChannel.open();
```

---

(2) 设置 `SocketChannel` 为非阻塞模式，同时设置客户端连接的 TCP 参数，示例代码如下：

---

```
clientChannel.configureBlocking(false);
socket.setReuseAddress(true);
socket.setReceiveBufferSize(BUFFER_SIZE);
socket.setSendBufferSize(BUFFER_SIZE);
```

---

(3) 异步连接服务端，示例代码如下：

---

```
boolean connected=clientChannel.connect(new InetSocketAddress("ip",port));
```

---

(4) 判断是否连接成功，如果连接成功，则直接注册读状态位到多路复用器中，如果当前没有连接成功（异步连接返回 `false`），说明客户端已经发送 `sync` 包，服务端没有返回 `ack` 包，物理链路还没有建立，示例代码如下：

---

```
if (connected)
{
    clientChannel.register( selector, SelectionKey.OP_READ, ioHandler);
}
else
{
    clientChannel.register( selector, SelectionKey.OP_CONNECT, ioHandler);
}
```

---

(5) 向 `Reactor` 线程的多路复用器注册 `OP_CONNECT` 事件，监听服务端的 TCP ACK 应答，示例代码如下：

---

```
clientChannel.register( selector, SelectionKey.OP_CONNECT, ioHandler);
```

---

(6) 创建 Reactor 线程，创建多路复用器并启动线程，示例代码如下：

---

```
Selector selector = Selector.open();
New Thread(new ReactorTask()).start();
```

---

(7) 多路复用器在线程 run 方法的无限循环体内轮询准备就绪的 Key，示例代码如下：

---

```
int num = selector.select();
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();
while (it.hasNext()) {
    SelectionKey key = (SelectionKey)it.next();
    // 处理 I/O 事件
}
```

---

(8) 接收 Connect 事件进行处理，示例代码如下：

---

```
if (key.isConnectable())
    //handlerConnect();
```

---

(9) 判断连接结果，如果连接成功，注册读事件到多路复用器，示例代码如下：

---

```
if (channel.finishConnect())
    registerRead();
```

---

(10) 注册读事件到多路复用器，示例代码如下：

---

```
clientChannel.register( selector, SelectionKey.OP_READ, ioHandler);
```

---

(11) 异步读客户端请求消息到缓冲区，示例代码如下：

---

```
int readNumber = channel.read(receivedBuffer);
```

---

(12) 对 ByteBuffer 进行编解码，如果有半包消息接收缓冲区 Reset，继续读取后续的报文，将解码成功的消息封装成 Task，投递到业务线程池中，进行业务逻辑编排，示例代码如下：

---

```
Object message = null;
```

---



```
while(buffer.hasRemain())
{
    byteBuffer.mark();
    Object message = decode(byteBuffer);
    if (message == null)
    {
        byteBuffer.reset();
        break;
    }
    messageList.add(message );
}
if (!byteBuffer.hasRemain())
byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
        handlerTask(messageE);
}
```

---

(13) 将 POJO 对象 encode 成 ByteBuffer，调用 SocketChannel 的异步 write 接口，将消息异步发送给客户端，示例代码如下：

---

```
socketChannel.write(buffer);
```

---

需要重点学习和掌握的类库包括 SocketChannel、SelectionKey、Selector 和 ByteBuffer。

## 2.2.2 Netty 客户端创建原理分析

Bootstrap 是 Socket 客户端创建工具类，用户通过 Bootstrap 可以方便地创建 Netty 的客户端并发起异步 TCP 连接操作，Netty 客户端创建流程如图 2-10 所示。

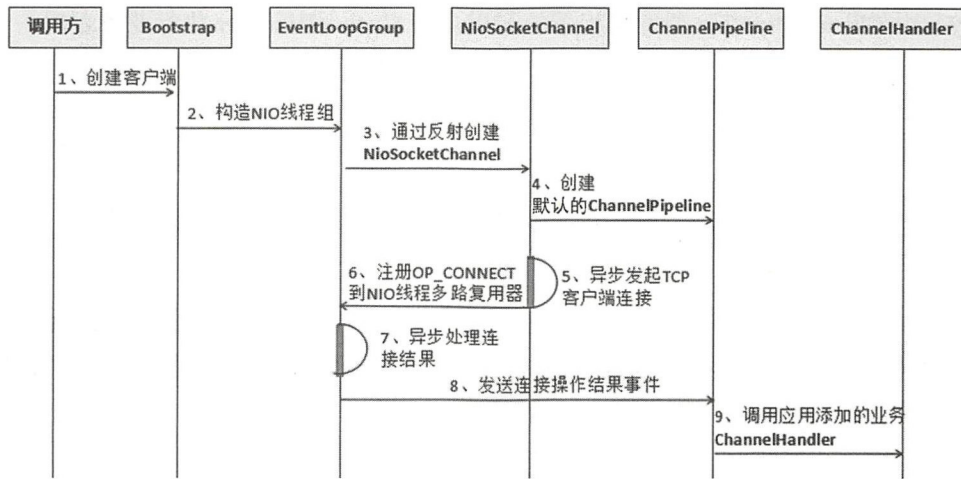


图 2-10 Netty 客户端创建流程

Netty 客户端创建流程说明如下。

(1) 用户线程创建 Bootstrap 实例，通过 API 设置创建客户端相关的参数，异步发起客户端连接。

(2) 创建处理客户端连接、I/O 读写的 Reactor 线程组 NioEventLoopGroup，可以通过构造函数指定 I/O 线程的个数，默认为 CPU 内核数的 2 倍。

(3) 通过 Bootstrap 的 ChannelFactory 和用户指定的 Channel 类型创建用于客户端连接的 NioSocketChannel，它的功能类似于 JDK NIO 类库提供的 SocketChannel。

(4) 创建默认的 ChannelPipeline，用于调度和执行网络事件。

(5) 异步发起 TCP 客户端连接，判断连接是否成功，如果成功，则直接将 NioSocketChannel 注册到多路复用器上，监听读操作位，用于数据报读取和消息发送；如果没有立即连接成功，则注册连接监听位到多路复用器，等待连接结果。

(6) 注册对应的网络监听状态位到多路复用器。

(7) 由多路复用器在 I/O 现场轮询各 Channel，处理连接结果。

(8) 如果连接成功，设置 Future 结果，发送连接成功事件，触发 ChannelPipeline 执行。

(9) 由 ChannelPipeline 调度执行系统和用户的 ChannelHandler 执行业务逻辑。

### 2.2.3 Bootstrap 工具类源码分析

Bootstrap 是 Netty 提供的客户端连接工具类，主要用于简化客户端的创建，下面对它的常用功能进行介绍。

设置 EventLoopGroup 线程池：通常多个连接会共享同一个 EventLoopGroup，EventLoopGroup 的大小默认为 CPU 内核数的 2 倍，也可以根据业务实际情况进行调整。如果性能和连接数都要求不高，建议设置为 1。

TCP 参数设置接口：无论是异步 NIO，还是同步 BIO，创建客户端套接字的时候通常都会设置连接参数，例如接收和发送缓冲区大小、连接超时时间等。Bootstrap 也提供了客户端 TCP 参数设置接口，代码如下（AbstractBootstrap 类）：

---

```
public <T> B option(ChannelOption<T> option, T value) {
    if (option == null) {
        throw new NullPointerException("option");
    }
    if (value == null) {
        synchronized (options) {
            options.remove(option);
        }
    } else {
        synchronized (options) {
            options.put(option, value);
        }
    }
    return self();
}
```

---

在实际项目中，通常需要设置的参数包括 TCP\_NODELAY、SO\_RCVBUF、SO\_SNDBUF、SO\_REUSEADDR、SO\_BACKLOG 和 SO\_LINGER。

Netty 提供的主要 TCP 参数如下。

(1) SO\_TIMEOUT：控制读取操作将阻塞多少毫秒。如果返回值为 0，计时器就被禁

止，该线程将无限期阻塞。

(2) `SO_SNDBUF`: 套接字使用的发送缓冲区大小。

(3) `SO_RCVBUF`: 套接字使用的接收缓冲区大小。

(4) `SO_REUSEADDR`: 决定当网络上仍然有数据向旧的 `ServerSocket` 传输数据时，是否允许新的 `ServerSocket` 绑定到与旧的 `ServerSocket` 同样的端口上。`SO_REUSEADDR` 选项的默认值与操作系统有关，在某些操作系统中，允许重用端口，而在某些其他操作系统中，不允许重用端口。

(5) `CONNECT_TIMEOUT_MILLIS`: 客户端连接超时时间，由于 NIO 原生的客户端并不提供设置连接超时的接口，因此 Netty 采用自定义连接超时定时器负责检测是否超时和进行超时控制。

(6) `TCP_NODELAY`: 激活或禁止 `TCP_NODELAY` 套接字选项，它决定是否使用 Nagle 算法。如果是时延敏感型的应用，建议关闭 Nagle 算法。

**Channel 接口**: 用于指定客户端使用的 Channel 接口，对于 TCP 客户端连接，默认使用 `NioSocketChannel`。`BootstrapChannelFactory` 利用 `channelClass` 类型信息，通过反射机制创建 `NioSocketChannel` 对象。

**添加和设置应用 ChannelHandler 接口**: `Bootstrap` 为了简化 `ChannelHandler` 的编排，提供了 `ChannelInitializer`，它继承了 `ChannelHandlerAdapter`，TCP 链路注册成功后，调用 `initChannel` 接口，用于设置用户 `ChannelHandler`，代码如下（`ChannelInitializer` 类）：

---

```
private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
    if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) {
        try {
            initChannel((C) ctx.channel());
        } catch (Throwable cause) {
            exceptionCaught(ctx, cause);
        } finally {
            remove(ctx);
        }
    }
    return true;
}
```

```
    }  
    return false;  
}
```

---

它的功能主要有两个。

(1) 执行客户端初始化时应用实现的 `initChannel(SocketChannel ch)`，将应用自定义的 `ChannelHandler` 添加到 `ChannelPipeline` 中。

(2) 将 `Bootstrap` 注册到 `ChannelPipeline` 用于初始化应用 `ChannelHandler` 的 `ChannelInitializer` 删除掉，完成应用 `ChannelPipeline` 的初始化工作。

最后一个重要接口就是调用 `Connect` 连接服务端，强烈建议采用异步的方式调用，即获取 `ChannelFuture` 后注册监听器，异步处理连接操作结果，不要阻塞调用方的线程。

## 2.3 总结

---

本章分析了一个生产环境 Netty 客户端连接池资源泄漏案例，详细讲解了 Netty 客户端创建的流程和工作原理，以及在实际项目中如何正确地实现连接池，避免发生并发安全和资源不当释放等问题。



## 第 3 章

---

# Netty 内存池泄漏疑云案例

为了提升消息接收和发送性能，Netty 针对 `ByteBuf` 的申请和释放采用池化技术，通过 `PooledByteBufAllocator` 可以创建基于内存池分配的 `ByteBuf` 对象，这样就避免了每次消息读写都申请和释放 `ByteBuf`。由于 `ByteBuf` 涉及 `byte[]` 数组的创建和销毁，对于性能要求苛刻的系统而言，重用 `ByteBuf` 带来的性能收益是非常可观的。

内存池是一把双刃剑，如果使用不当，很容易带来内存泄漏和内存非法引用等问题，另外，除了内存池，Netty 同时也支持非池化的 `ByteBuf`，多种类型的 `ByteBuf` 功能存在一些差异，使用不当很容易带来各种问题。

本章通过对一个内存池使用不当导致的内存泄漏案例做分析，详细介绍 `ByteBuf` 的申请和释放策略，以及 Netty 内存池的工作原理。

### 3.1 Netty 内存池泄漏问题

业务路由分发模块使用 Netty 作为通信框架，负责协议消息的接入和路由转发，在功能测试时没有发现问题，转性能测试之后，运行一段时间就发现内存分配异常，服务端无法接收请求消息，系统吞吐量降为 0。

### 3.1.1 路由转发服务代码

作为案例示例，对业务服务路由转发代码进行简化，以方便分析：

```
public class RouterServerHandler extends ChannelInboundHandlerAdapter {
    static ExecutorService executorService = Executors.newSingleThreadExecutor();
    PooledByteBufAllocator allocator = new PooledByteBufAllocator(false);
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf reqMsg = (ByteBuf)msg;
        byte [] body = new byte[reqMsg.readableBytes()];
        executorService.execute(()->
        {
            //解析请求消息，做路由转发，代码省略
            //转发成功，返回响应给客户端
            ByteBuf respMsg = allocator.heapBuffer(body.length);
            respMsg.writeBytes(body); //作为示例，简化处理，将请求返回
            ctx.writeAndFlush(respMsg);
        });
    }
    //后续代码省略
}
```

进行一段时间的性能测试之后，日志中出现异常，进程内存不断飙升，怀疑存在内存泄漏问题，如图 3-1 所示。

```
io.netty.util.internal.OutOfDirectMemoryError: failed to allocate 16777216 byte(s) of direct memory (used: 905969664, max: 913833984)
    at io.netty.util.internal.PlatformDependent.incrementMemoryCounter(PlatformDependent.java:640)
    at io.netty.util.internal.PlatformDependent.allocateDirectNoCleaner(PlatformDependent.java:594)
    at io.netty.buffer.PoolArena$DirectArena.allocateDirect(PoolArena.java:764)
    at io.netty.buffer.PoolArena$DirectArena.newChunk(PoolArena.java:740)
    at io.netty.buffer.PoolArena.allocateNormal(PoolArena.java:244)
    at io.netty.buffer.PoolArena.allocate(PoolArena.java:214)
    at io.netty.buffer.PoolArena.allocate(PoolArena.java:146)
    at io.netty.buffer.PoolArena$PooledByteBufAllocator.newDirectBuffer(PooledByteBufAllocator.java:324)
    at io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.java:185)
    at io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.java:176)
    at io.netty.buffer.AbstractByteBufAllocator.ioBuffer(AbstractByteBufAllocator.java:137)
    at io.netty.channel.DefaultMaxMessagesRecvByteBufAllocator$MaxMessageHandle.allocate(DefaultMaxMessagesRecvByteBufAllocator.java:114)
    at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:147)
    at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:646)
    at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:581)
    at io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:498)
```

图 3-1 性能测试异常日志

### 3.1.2 响应消息内存释放玄机

对业务 ByteBuf 申请相关代码进行排查,发现响应消息由业务线程创建,但是却没有主动释放,因此怀疑是响应消息没有释放导致的内存泄漏。因为响应消息使用的是 PooledHeapByteBuf,如果发生内存泄漏,利用堆内存监控就可以找到泄漏点,通过 Java VisualVM 工具观察堆内存占用趋势,并没有发现堆内存发生泄漏,如图 3-2 所示。

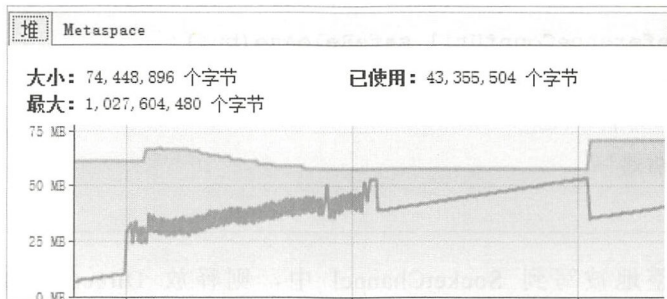


图 3-2 业务堆内存监控数据

对内存做快照,查看在性能压测过程中响应消息 PooledUnsafeHeapByteBuf 的实例个数,如图 3-3 所示,响应消息对象个数和内存占用都很少,排除内存泄漏嫌疑。

Profiler 快照				
类名 - 活动的分配对象	活动字节 [%]	活动字节 ▲	活动对象	年代数
io.netty.buffer.PooledUnsafeHeapByteBuf		12,240 B (2.4%)	153 (3.1%)	1

图 3-3 业务堆内存快照

业务从内存池中申请了 ByteBuf,但是却没有主动释放它,最后也没有发生内存泄漏,这究竟是什么原因呢?通过对 Netty 源码的分析,我们破解了其中的玄机。原来调用 ctx.writeAndFlush(respMsg)方法时,当消息发送完成,Netty 框架会主动帮助应用释放内存,内存的释放分为如下两种场景。

(1)如果是堆内存(PooledHeapByteBuf),则将 HeapByteBuffer 转换成 DirectByteBuffer,并释放 PooledHeapByteBuf 到内存池,代码如下 (AbstractNioChannel 类):

```
protected final ByteBuf newDirectBuffer(ByteBuf buf) {
    final int readableBytes = buf.readableBytes();
    if (readableBytes == 0) {
```

```

        ReferenceCountUtil.safeRelease(buf);
        return Unpooled.EMPTY_BUFFER;
    }

    final ByteBufAllocator alloc = alloc();
    if (alloc.isDirectBufferPooled()) {
        ByteBuf directBuf = alloc.directBuffer(readableBytes);
        directBuf.writeBytes(buf, buf.readerIndex(), readableBytes);
        ReferenceCountUtil.safeRelease(buf);
        return directBuf;
    }
}
//后续代码省略
}

```

---

如果消息完整地写到 SocketChannel 中，则释放 DirectByteBuffer，代码如下 (ChannelOutboundBuffer)：

```

public boolean remove() {
    Entry e = flushedEntry;
    if (e == null) {
        clearNioBuffers();
        return false;
    }
    Object msg = e.msg;
    ChannelPromise promise = e.promise;
    int size = e.pendingSize;
    removeEntry(e);
    if (!e.cancelled) {
        ReferenceCountUtil.safeRelease(msg);
        safeSuccess(promise);
        decrementPendingOutboundBytes(size, false, true);
    }
}
//后续代码省略
}

```

---

对 Netty 源码进行断点调试，验证上述分析。

断点 1：在响应消息发送处设置断点，获取到的 `PooledUnsafeHeapByteBuf` 实例的 ID 为 1506，如图 3-4 所示。

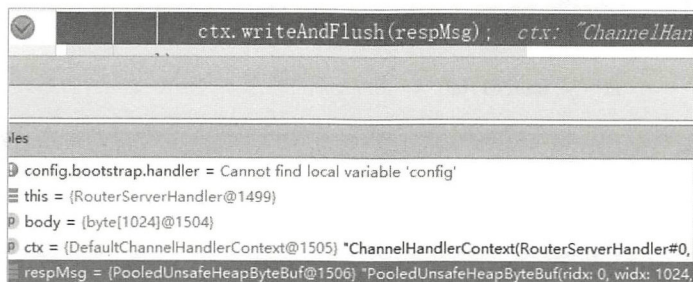


图 3-4 在响应消息发送处设置断点

断点 2：在 `HeapByteBuffer` 转换成 `DirectByteBuffer` 处设置断点，发现实例 ID 为 1506 的 `PooledUnsafeHeapByteBuf` 被释放，如图 3-5 所示。

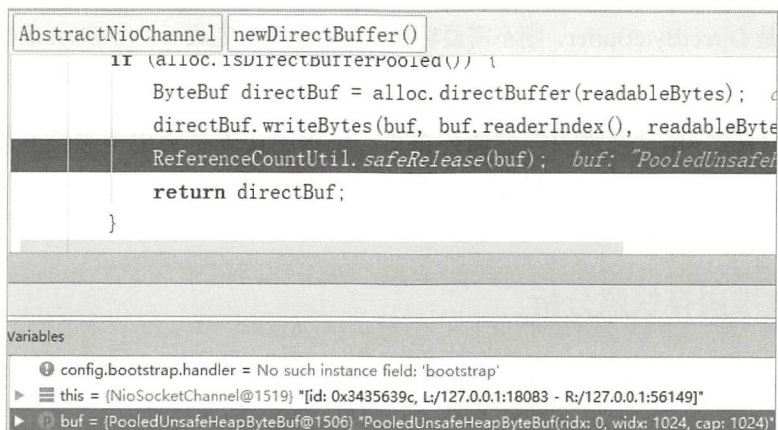


图 3-5 在响应消息释放处设置断点

断点 3：转换之后待发送的响应消息 `PooledUnsafeDirectByteBuf` 实例的 ID 为 1527，如图 3-6 所示。

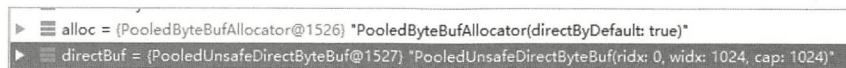


图 3-6 在响应消息转换处设置断点

断点 4：在响应消息发送完成后，实例 ID 为 1527 的 `PooledUnsafeDirectByteBuf` 被释



放到内存池中，如图 3-7 所示。

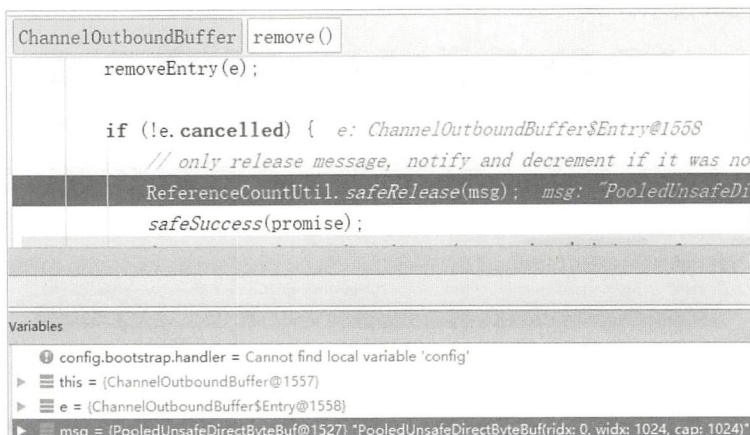


图 3-7 在转换之后的响应消息释放处设置断点

(2) 如果是 DirectByteBuffer，则不需要转换，在消息发送完成后，由 ChannelOutboundBuffer 的 remove() 负责释放。

通过源码解读、调试及堆内存的监控分析，可以确认不是响应消息没有主动释放导致的内存泄漏，需要 Dump 内存做进一步定位。

### 3.1.3 采集堆内存快照分析

执行 jmap 命令，Dump 应用内存堆栈，如图 3-8 所示。

```

C:\nio\netty\heap>jmap -dump:format=b,file=router_memory.hprof 1040
Dumping heap to C:\nio\netty\heap\router_memory.hprof ...
Heap dump file created
  
```

图 3-8 Dump 应用内存堆栈的命令

通过 MemoryAnalyzer 工具对内存堆栈进行分析，寻找内存泄漏点，如图 3-9 所示。

从图 3-9 可以看出，内存泄漏点是 Netty 内存池对象 PoolChunk，由于请求和响应消息内存分配都来自 PoolChunk，暂时还不确认是请求还是响应消息导致的问题。进一步对代码进行分析，发现响应消息使用的是堆内存 HeapByteBuffer，请求消息使用的是 DirectByteBuffer，由于 Dump 出来的是堆内存，如果是堆内存泄漏，Dump 出来的内存文



件应该包含大量的 `PooledHeapByteBuf`，实际上并没有，因此可以确认系统发生了堆外内存泄漏，即请求消息没有被释放或者没有被及时释放导致的内存泄漏。

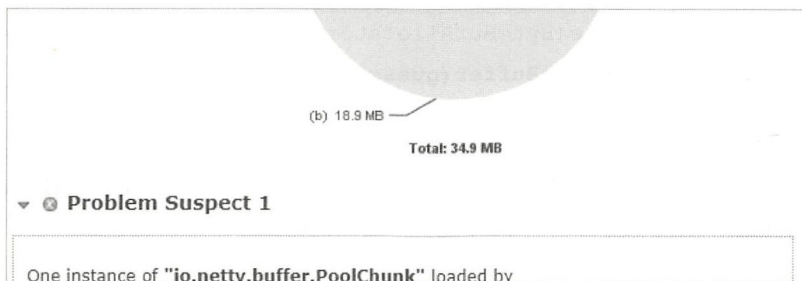


图 3-9 寻找内存泄漏点

对请求消息的内存分配进行分析，发现在 `NioByteUnsafe` 的 `read` 方法中申请了内存，代码如下（`NioByteUnsafe`）：

```
public final void read() {  
    final ChannelConfig config = config();  
    if (shouldBreakReadReady(config)) {  
        clearReadPending();  
        return;  
    }  
    final ChannelPipeline pipeline = pipeline();  
    final ByteBufAllocator allocator = config.getAllocator();  
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();  
    allocHandle.reset(config);  
  
    ByteBuf byteBuf = null;  
    boolean close = false;  
    try {  
        do {  
            byteBuf = allocHandle.allocate(allocator);  
            //代码省略  
        } while (true);  
    }  
}
```

继续对 `allocate` 方法进行分析，发现调用的是 `DefaultMaxMessagesRecvByteBuf-`



Allocator\$MaxMessageHandle 的 allocate 方法,代码如下(DefaultMaxMessagesRecvByteBufferAllocator):

---

```
public ByteBuffer allocate(ByteBufferAllocator alloc) {  
    return alloc.ioBuffer(guess());  
}
```

---

alloc.ioBuffer 方法最终会调用 PooledByteBufferAllocator 的 newDirectBuffer 方法创建 PooledDirectByteBuffer 对象。

请求 ByteBuffer 的创建分析完,继续分析它的释放操作,由于业务的 RouterServerHandler 继承自 ChannelInboundHandlerAdapter, 它的 channelRead(ChannelHandlerContext ctx, Object msg)方法执行完成, ChannelHandler 的执行就结束了, 代码示例如下:

---

```
@Override  
public void channelRead(ChannelHandlerContext ctx, Object msg) {  
    ByteBuffer reqMsg = (ByteBuffer)msg;  
    byte [] body = new byte[reqMsg.readableBytes()];  
    executorService.execute(()->  
    {  
        //解析请求消息,做路由转发,代码省略  
        //转发成功,返回响应给客户端  
        ByteBuffer respMsg = allocator.heapBuffer(body.length);  
        respMsg.writeBytes(body); //作为示例,简化处理,将请求返回  
        ctx.writeAndFlush(respMsg);  
    });  
    //后续代码省略
```

---

通过代码分析发现, 请求 ByteBuffer 被 Netty 框架申请后竟然没有被释放, 为了验证分析, 在业务代码中调用 ReferenceCountUtil 的 release 方法进行内存释放操作, 代码修改如下:

---

```
@Override  
public void channelRead(ChannelHandlerContext ctx, Object msg) {  
    ByteBuffer reqMsg = (ByteBuffer)msg;
```

---

```
byte [] body = new byte[reqMsg.readableBytes()];  
ReferenceCountUtil.release(reqMsg);  
  
//后续代码省略
```

修改之后继续进行压测，发现系统运行平稳，没有发生 OOM 异常。对内存活动对象进行排序，没有再发现大量的 PoolChunk 对象，内存泄漏问题解决，问题修复之后的内存快照如图 3-10 所示。

类名 - 活动的分配对象	活动字节 [%]	活动字节	活动对象	年代数
byte[]		44,720 B (36.2%)	40 (2%)	10
java.lang.Object[]		10,894 B (8.8%)	194 (9.7%)	105
java.util.HashMap		8,832 B (7.2%)	184 (9.2%)	100
java.util.Vector		5,824 B (4.7%)	182 (9.1%)	110
java.util.Hashtable\$Entry[]		5,312 B (4.3%)	83 (4.1%)	77
java.util.Hashtable		4,464 B (3.6%)	93 (4.6%)	88
io.netty.util.Recycler\$DefaultHandle		4,448 B (3.6%)	139 (6.9%)	27
java.nio.DirectByteBuffer		4,224 B (3.4%)	66 (3.3%)	8
java.security.ProtectionDomain		3,800 B (3.1%)	95 (4.7%)	78
java.security.CodeSource		2,912 B (2.4%)	91 (4.5%)	84

图 3-10 问题修复之后的内存快照

### 3.1.4 ByteBuf 申请和释放的理解误区

有一种说法认为 Netty 框架分配的 ByteBuf 框架会自动释放，业务不需要释放；业务创建的 ByteBuf 则需要自己释放，Netty 框架不会释放。

通过前面的案例分析和验证，我们可以看出这个观点是错误的。为了在实际项目中更好地管理 ByteBuf，下面我们分 4 种场景进行说明。

#### 1. 基于内存池的请求 ByteBuf

这类 ByteBuf 主要包括 PooledDirectByteBuffer 和 PooledHeapByteBuffer，它由 Netty 的 NioEventLoop 线程在处理 Channel 的读操作时分配，需要在业务 ChannelInboundHandler 处理完请求消息之后释放（通常在解码之后），它的释放有两种策略。

**策略 1** 业务 ChannelInboundHandler 继承自 SimpleChannelInboundHandler，实现它的抽象方法 channelRead0(ChannelHandlerContext ctx, I msg)，ByteBuffer 的释放业务不用关心，由 SimpleChannelInboundHandler 负责释放，相关代码如下（SimpleChannelInboundHandler）：

```
@Override  
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
```



## Netty 进阶之路：跟着案例学 Netty

```
Exception {
    boolean release = true;
    try {
        if (acceptInboundMessage(msg)) {
            I imsg = (I) msg;
            channelRead0(ctx, imsg);
        } else {
            release = false;
            ctx.fireChannelRead(msg);
        }
    } finally {
        if (autoRelease && release) {
            ReferenceCountUtil.release(msg);
        }
    }
}
```

如果当前业务 `ChannelInboundHandler` 需要执行，则调用 `channelRead0` 之后执行 `ReferenceCountUtil.release(msg)` 释放当前请求消息。如果没有匹配上需要继续执行后续的 `ChannelInboundHandler`，则不释放当前请求消息，调用 `ctx.fireChannelRead(msg)` 驱动 `ChannelPipeline` 继续执行。

对案例中的问题代码进行修改，继承自 `SimpleChannelInboundHandler`，即便业务不释放请求的 `ByteBuf` 对象，依然不会发生内存泄漏，修改之后的代码如下（`RouterServerHandlerV2`）：

```
public class RouterServerHandlerV2 extends SimpleChannelInboundHandler
<ByteBuf> {
    //代码省略
    @Override
    public void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) {
        byte [] body = new byte[msg.readableBytes()];
        executorService.execute(() ->
        {
            //解析请求消息，做路由转发，代码省略
        }
    }
}
```





```
//转发成功，返回响应给客户端
ByteBuf respMsg = allocator.heapBuffer(body.length);
respMsg.writeBytes(body); //作为示例，简化处理，将请求返回
ctx.writeAndFlush(respMsg);
});
}
```

对修改之后的代码做性能测试，发现内存占用平稳，无内存泄漏问题，验证了之前的分析结论。

**策略 2** 在业务 `ChannelInboundHandler` 中调用 `ctx.fireChannelRead(msg)` 方法，让请求消息继续向后执行，直到调用 `DefaultChannelPipeline` 的内部类 `TailContext`，由它来负责释放请求消息，代码如下（`TailContext`）：

```
protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug(
            "Discarded inbound message {} that reached at the tail of
the pipeline. " +
            "Please check your pipeline configuration.", msg);
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

## 2. 基于非内存池的请求 ByteBuf

如果业务使用非内存池模式覆盖 Netty 默认的内存池模式创建请求 `ByteBuf`，例如通过如下代码修改内存申请策略为 `Unpooled`：

```
//代码省略
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline p = ch.pipeline();
        ch.config().setAllocator(UnpooledByteBufAllocator.DEFAULT);
    }
})
```



```
        p.addLast(new RouterServerHandler());  
    }  
    });  
}
```

---

也需要按照内存池的方式释放内存。

### 3. 基于内存池的响应 ByteBuf

根据之前的分析，只要调用了 `writeAndFlush` 或者 `flush` 方法，在消息发送完成后都会由 Netty 框架进行内存释放，业务不需要主动释放内存。

### 4. 基于非内存池的响应 ByteBuf

无论是基于内存池还是非内存池分配的 `ByteBuf`，如果是堆内存，则将堆内存转换成堆外内存，然后释放 `HeapByteBuffer`，待消息发送完成，再释放转换后的 `DirectByteBuffer`；如果是 `DirectByteBuffer`，则不需要转换，待消息发送完成之后释放。因此对于需要发送的响应 `ByteBuf`，由业务创建，但是不需要由业务来释放。

## 3.2 Netty 内存池工作机制

作为高性能的 NIO 通信框架，Netty 在 RPC、分布式服务框架、MQ、大数据等领域得到了广泛的应用，性能优势是 Netty 的核心竞争力之一，自从 Netty 4.X 引入内存池机制后，Netty 默认采用内存池模式创建 `ByteBuf` 对象，性能得到了很大提升，GC 压力也得到了很大缓解。

### 3.2.1 内存池的性能优势

通过 `Pooled` 和 `Unpooled` 方式创建 `ByteBuf`，模拟请求消息的创建和释放，采用非内存池方式创建的 `ByteBuf` 性能测试代码如下：

---

```
static void unPoolTest()  
{
```



```
//非内存池模式
long beginTime = System.currentTimeMillis();
ByteBuf buf = null;
int maxTimes = 100000000;
for(int i = 0; i < maxTimes; i++)
{
    buf = Unpooled.buffer(10 * 1024);
    buf.release();
}
System.out.println("Execute " + maxTimes + " times cost time : "
    + (System.currentTimeMillis() - beginTime));
}
```

查看性能测试结果，如图 3-11 所示，执行 1 亿次耗时 126s。

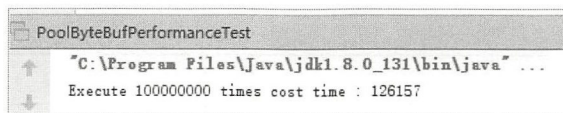


图 3-11 通过非内存池模式创建的 ByteBuf 性能测试结果

查看 GC 数据，如图 3-12 所示，执行 GC 3038 次，耗时 2.34s。

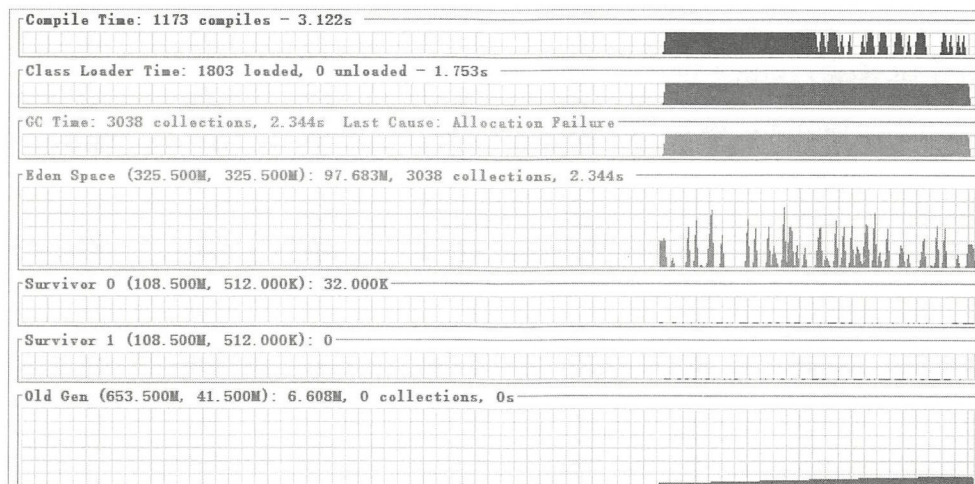


图 3-12 非内存池模式 GC 数据



通过内存池模式创建 ByteBuf，性能测试代码如下：

```
static void poolTest()
{
    //内存池模式
    PooledByteBufAllocator allocator = new PooledByteBufAllocator(false);
    long beginTime = System.currentTimeMillis();
    ByteBuf buf = null;
    int maxTimes = 100000000;
    for(int i = 0; i < maxTimes; i++)
    {
        buf = allocator.heapBuffer(10 * 1024);
        buf.release();
    }
    System.out.println("Execute " + maxTimes + " times cost time : "
        + (System.currentTimeMillis() - beginTime));
}
```

查看性能测试结果，如图 3-13 所示。

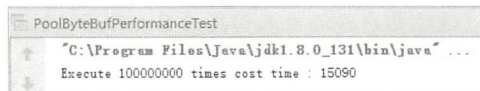


图 3-13 通过内存池模式创建的 ByteBuf 性能测试结果

查看 GC 数据，如图 3-14 所示，执行 GC 32 次，耗时 37.6ms。

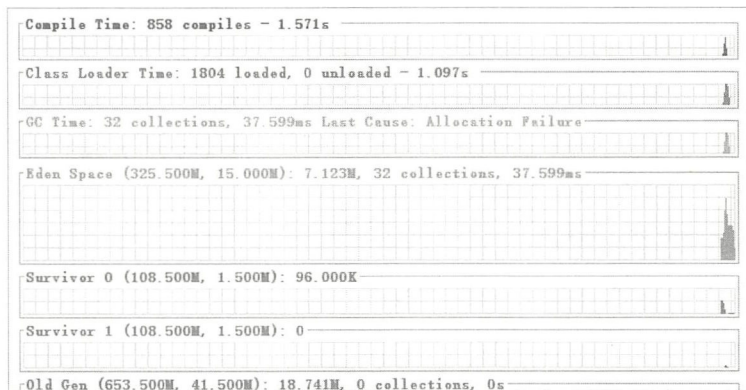


图 3-14 内存池模式 GC 数据



从以上简单的性能测试对比发现,采用内存池模式创建的 `ByteBuf` 性能比传统非池化方式高 8 倍多。

在 API Gateway、RPC 和流式处理框架中,请求和响应消息往往是“朝生熄灭”的,特别是频繁地申请和释放大块 `byte` 数组,会加重 GC 的负担及加大 CPU 资源占用率,通过内存池技术重用这些临时对象,可以降低 GC 频次和减少耗时,同时提升系统的吞吐量。

### 3.2.2 内存池工作原理分析

Netty 的内存池整体上参照 `jemalloc` 实现,主要数据结构如下。

(1) `PooledArena`: 代表内存中一大块连续的区域, `PoolArena` 由多个 `Chunk` 组成,每个 `Chunk` 由多个 `Page` 组成。为了提升并发性能,内存池中包含一组 `PooledArena`。

(2) `PoolChunk`: 用来组织和管理多个 `Page` 的内存分配和释放,默认为 16MB。

(3) `PoolSubpage`: 对于小于一个 `Page` 的内存, Netty 在 `Page` 中完成分配。每个 `Page` 会被切分成大小相等的多个存储块,存储块的大小由第一次申请的内存块大小决定。假如一个 `Page` 是 8 字节,如果第一次申请的块大小是 4 字节,那么这个 `Page` 就包含两个存储块;如果第一次申请的块大小是 8 字节,那么这个 `Page` 就被分成一个存储块。一个 `Page` 只能用于分配与第一次申请时大小相同的内存,比如,一个 4 字节的 `Page`,如果第一次分配了 1 字节的内存,那么后面这个 `Page` 只能继续分配 1 字节的内存,如果有一个申请 2 字节内存的请求,就需要在新的 `Page` 中进行分配。

内存池的内存分配从 `PooledArena` 开始,一个 `PooledArena` 包含多个 `Chunk` (`PoolChunk`), `Chunk` 具体负责内存的分配和回收。每个 `Chunk` 包含多个 `Page` (`PoolSubpage`),每个 `Page` 由大小相等的块 (`Region`) 组成,每个 `Page` 块大小由第一次从 `Page` 申请的内存大小决定,某个 `Page` 中的块大小是相等的。`PoolChunk` 默认为 16MB,包含 2048 个 `Page`,每个 `Page` 为 8KB。

内存分配策略:通过 `PooledByteBufAllocator` 申请内存时,首先从 `PoolThreadLocalCache` 中获取与线程绑定的缓存池 `PoolThreadCache`,如果不存在线程私有缓存池,则轮询分配一个 `Arena` 数组中的 `PooledArena`,创建一个新的 `PoolThreadCache` 作为缓存池使用。

`PooledArena` 在进行内存分配时对预分配的内存容量做判断,分为如下几种场景:



- (1) 需要分配的内存小于 `PageSize` 时，分配 `tiny` 或者 `small` 内存。
- (2) 需要分配的内存介于 `PageSize` 和 `ChunkSize` 之间时，则分配 `normal` 内存。
- (3) 需要分配的内存大于 `ChunkSize` 时，则分配 `huge` 内存（非池化内存）。

Netty 内存池将内存分为几类: tiny (小于 512B)、small (大于等于 512B, 小于 8KB)、normal (大于等于 8KB, 小于等于 16MB) 和 huge (大于 16MB), 系统根据需要申请的内存大小选择合适的 MemoryRegionCache。

在 PooledArena 中创建 PoolChunk 后，调用 PoolChunk 的 allocate() 方法进行真正的内存分配：PoolChunk 通过二叉树记录每个 PoolSubpage 的分配情况，如图 3-15 所示。

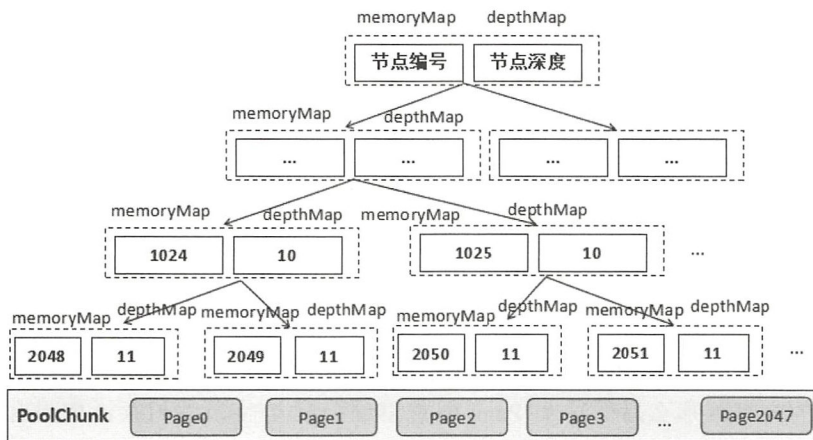


图 3-15 PoolChunk 二叉树

PoolChunk 用 memoryMap 和 depthMap 来表示二叉树，其中 memoryMap 存放的是 PoolSubpage 的分配信息，depthMap 存放的是二叉树的深度。depthMap 初始化之后就不再变化，而 memoryMap 则随着 PoolSubpage 的分配而改变。初始化时，memoryMap 和 depthMap 的取值相同。节点的分配情况有如下三种可能。

- (1) `memoryMap[id] = depthMap[id]`: 表示当前节点可分配内存。
- (2) `memoryMap[id] > depthMap[id]`: 表示当前节点至少一个子节点已经被分配, 无法分配满足该深度的内存, 但是可以分配更小一些的内存 (通过空闲的子节点)。
- (3) `memoryMap[id] = 最大深度 (默认为 11) + 1`: 表示当前节点下的所有子节点都

已经分配完，没有可用内存。

假设申请 4KB 的内存，内存分配过程如图 3-16 所示。

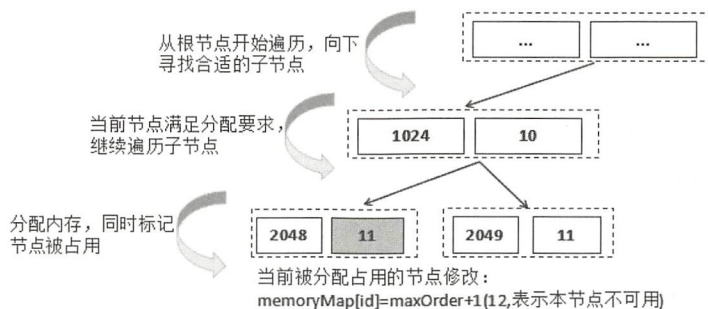


图 3-16 内存分配过程

节点被标记为被占用之后，依次向上遍历更新父节点，直到根节点。将父节点的  $\text{memoryMap}[\text{id}]$  位置信息修改为两个子节点中的较小值，如图 3-17 所示。

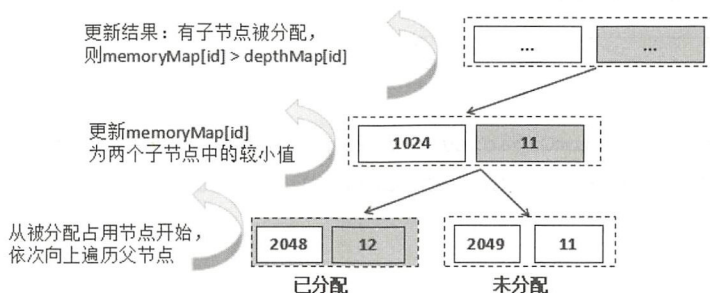


图 3-17 二叉树父节点更新

获取到合适的节点之后，根据节点 ID 计算并获取  $\text{subpageIdx}$ ，通过  $\text{subpageIdx}$  从  $\text{PoolSubpage}\langle T \rangle[] \text{ subpages}$  中获取可用的  $\text{PoolSubpage}$ 。如果  $\text{PoolSubpage}$  为空，则新建  $\text{PoolSubpage}$  对象，并将其加入  $\text{PoolChunk}$  的  $\text{PoolSubpage}\langle T \rangle[] \text{ subpages}$  中，就可以参与后续的内存分配了。如果  $\text{PoolSubpage}$  不为空，说明是被使用完之后释放到了内存池中，重新初始化  $\text{PoolSubpage}$ ，并将其加入内存池的双向链表中，参与内存分配。

调用  $\text{PoolSubpage}$  的  $\text{allocate}$  方法，返回代表  $\text{PoolSubpage}$  块存储区域的占用情况结果 (long 类型：高 32 位表示  $\text{subPage}$  中分配的位置，低 32 位表示二叉树中分配的节点)， $\text{PoolArena}$  根据上述信息调用  $\text{PoolChunk}$  的  $\text{initBuf}$  方法完成  $\text{PooledByteBuf}$  的内存分配。

### 3.2.3 内存池核心代码分析

PooledByteBufAllocator 的 newDirectBuffer 方法，首先从线程缓存中获取 PoolArena，示例代码如下：

---

```
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    PoolThreadCache cache = threadCache.get();
    PoolArena<ByteBuf> directArena = cache.directArena;
    final ByteBuf buf;
    if (directArena != null) {
        buf = directArena.allocate(cache, initialCapacity, maxCapacity);
    }
    //后续代码省略
}
```

---

调用 PoolArena 的 allocate 方法分配内存，代码如下：

---

```
private void allocate(PoolThreadCache cache, PooledByteBuf<T> buf, final int reqCapacity) {
    final int normCapacity = normalizeCapacity(reqCapacity);
    if (isTinyOrSmall(normCapacity)) { // capacity < pageSize
        int tableIdx;
        PoolSubpage<T>[] table;
        boolean tiny = isTiny(normCapacity);
        if (tiny) { // < 512
            //此处代码省略
        } else {
            if (cache.allocateSmall(this, buf, reqCapacity, normCapacity)) {
                //此处代码省略
            }
        }
    }
    if (normCapacity <= chunkSize) {
        if (cache.allocateNormal(this, buf, reqCapacity, normCapacity)) {
            return;
        }
    }
}
```

---

```

    }
    synchronized (this) {
        allocateNormal(buf, reqCapacity, normCapacity);
        ++allocationsNormal;
    }
} else {
    allocateHuge(buf, reqCapacity);
}
}
}
//后续代码省略
}

```

根据申请的内存容量选择合适的 MemoryRegionCache, 然后调用 PoolChunk 的 allocate 方法分配内存, PoolChunk 的内存分配策略如下。

(1) 查找 PoolChunk 所在 PoolArena 的 PoolSubpage 头节点, 因为在内存分配过程中会修改 PoolSubpage 链表, 考虑到并发安全需要加锁, 代码如下 (PoolChunk):

```

private long allocateSubpage(int normCapacity) {
    PoolSubpage<T> head = arena.findSubpagePoolHead(normCapacity);
    synchronized (head) {
        //后续代码省略
    }
}

```

(2) 从二叉树中查找适合分配的节点, 获取节点 ID, 代码如下:

```

private int allocateNode(int d) {
    int id = 1;
    int initial = - (1 << d); // has last d bits = 0 and rest all = 1
    byte val = value(id);
    if (val > d) { //不可用
        return -1;
    }
    while (val < d || (id & initial) == 0) { // id & initial == 1 << d
        for all ids at depth d, for < d it is 0
        id <<= 1;
    }
}

```

```

        val = value(id);
        if (val > d) {
            id ^= 1;
            val = value(id);
        }
    } //后续代码省略
}

```

---

层层向下判断，当子节点满足分配条件时，深度加 1，进入子节点，先判断左节点是否满足分配要求，如果不满足则寻找同层级的右节点，直到找到符合要求的节点，获得节点 ID。

(3) 标记已分配的节点 ID 为不可用，并层层向上更新父节点的分配信息(memoryMap[id]):

---

```

private int allocateNode(int d) {
    //此处代码省略

    setValue(id, unusable); //标记为不可用
    updateParentsAlloc(id);
}

```

---

(4) 根据 nodeID 获取对应的 PoolSubpage，代码如下 (PoolChunk):

---

```

private long allocateSubpage(int normCapacity) {
    //此处代码省略

    int subpageIdx = subpageIdx(id);
    PoolSubpage<T> subpage = subpages[subpageIdx];
}

```

---

subpageIdx 方法用于获取节点的偏移索引，它对应于 PoolSubpage<T>[] subpages 的下标。

(5) 判断 PoolSubpage 是新创建的还是被释放重用的，如果是新建的，创建完成之后加入 PoolSubpage<T>[] subpages 中；如果是重用的，则重新初始化 PoolSubpage，更新 Page 的元数据信息（包括 elemSize 和 bitmap 等），并将更新之后的 PoolSubpage 加入内存池的双向链表中，代码如下：



---

```

private long allocateSubpage(int normCapacity) {
    //此处代码省略

    PoolSubpage<T> subpage = subpages[subpageIdx];
    if (subpage == null) {
        subpage = new PoolSubpage<T>(head, this, id, runOffset(id),
        pageSize, normCapacity);
        subpages[subpageIdx] = subpage;
    } else {
        subpage.init(head, normCapacity);
    } }

```

---

调用 PoolSubpage 的 allocate 方法, 返回 PoolSubpage 分配情况的位图索引 (低 32 位表示二叉树节点分配信息 memoryMapIdx):

---

```

private long allocateSubpage(int normCapacity) {
    long allocate() {
        if (elemSize == 0) {
            return toHandle(0);
        }
        if (numAvail == 0 || !doNotDestroy) {
            return -1;
        }
        final int bitmapIdx = getNextAvail();
        int q = bitmapIdx >>> 6;
        int r = bitmapIdx & 63;
        assert (bitmap[q] >>> r & 1) == 0;
        bitmap[q] |= 1L << r;
        if (-- numAvail == 0) {
            removeFromPool();
        }
        return toHandle(bitmapIdx);
    }
}

```

---

最后根据位图索引、需申请的内存容量等参数完成内存分配，代码如下(PoolChunk)：

---

```
void initBuf(PooledByteBuf<T> buf, long handle, int reqCapacity) {
    int memoryMapIdx = memoryMapIdx(handle);
    int bitmapIdx = bitmapIdx(handle);
    if (bitmapIdx == 0) {
        byte val = value(memoryMapIdx);
        assert val == unusable : String.valueOf(val);
        buf.init(this, handle, runOffset(memoryMapIdx) + offset,
reqCapacity, runLength(memoryMapIdx),
                arena.parent.threadCache());
    } else {
        initBufWithSubpage(buf, handle, bitmapIdx, reqCapacity);
    }
}
```

---

Netty 内存池的实现细节还是非常复杂的，限于篇幅及本章的侧重点等，不再做更详细的源码讲解（包括内存释放机制）。如果读者对 Netty 内存池底层实现细节比较感兴趣，可以通过“jemalloc 原理学习+Netty 源码调试”的方式，更深入地学习和掌握 Netty 内存池。对于普通的业务开发而言，重要的是熟悉 Netty 内存池的工作原理，以及常用内存分配相关 API 的使用与约束。

### 3.3 总结

Netty 内存池是一把双刃剑，使用得当会在很大程度上提升系统的性能，但是误用则会带来内存泄漏问题。从表面上看，只要遵循主动申请和释放原则即可，但是由于内存的申请和释放可能由 Netty 框架隐性完成，增加了内存管理的复杂性。

通过学习 Netty 收发消息的 ByteBuf 申请和释放机制，可以避免在项目中因误用 ByteBuf 而发生内存泄漏。在熟悉了 ByteBuf 的申请和释放机制后，通过对 Netty 内存池工作原理和关键源码的分析，读者可以更好地掌握 Netty 内存池的使用方法。

## 第 4 章

---

# ByteBuf 故障排查案例

应用在传输数据时，往往需要使用缓冲区，最常用的缓冲区就是 JDK NIO 类库提供的 `java.nio.Buffer`。由于 JDK 原生的 `Buffer` 存在一些缺点，Netty 提供了自己的 `ByteBuffer` 实现（`ByteBuf` 类）。

`ByteBuf` 种类繁多，支持的功能特性存在一些差异，如果使用不当，往往会出现功能异常或者性能问题。本章通过一个业务 `ByteBuf` 的故障排查案例，讲解在 Netty HTTP 协议栈中如何正确地使用 `ByteBuf`，以及 `ByteBuf` 的工作原理。

## 4.1 HTTP 协议栈 ByteBuf 使用问题

---

在使用 Netty 开发 Restful 应用时，我由于 `ByteBuf` 使用不当，曾遇到几个问题，现将问题的原因和定位过程做详细说明。

### 4.1.1 HTTP 响应 Body 获取异常

---

HTTP 客户端的代码示例如下（为了方便说明，简化业务代码，采用同步 HTTP 调用

方式做示例解说)：

---

```

public class HttpClient {
    private Channel channel;

    HttpClientHandler handler = new HttpClientHandler();

    private void connect(String host, int port) throws Exception {
        EventLoopGroup workerGroup = new NioEventLoopGroup(1);
        Bootstrap b = new Bootstrap();
        b.group(workerGroup);
        b.channel(NioSocketChannel.class);
        b.handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception {
                ch.pipeline().addLast(new HttpClientCodec());
                ch.pipeline().addLast(new HttpObjectAggregator(Short.MAX_
VALUE));

                ch.pipeline().addLast(handler);
            }
        });
        ChannelFuture f = b.connect(host, port).sync();
        channel = f.channel();
    }

    private HttpResponse blockSend(FullHttpRequest request) throws
InterruptedException, ExecutionException
    {
        request.headers().set(HttpHeaderNames.CONTENT_LENGTH,
request.content().readableBytes());

        DefaultPromise<HttpResponse> respPromise = new DefaultPromise
<HttpResponse>(channel.eventLoop());
        handler.setRespPromise(respPromise);
        channel.writeAndFlush(request);
        HttpResponse response = respPromise.get();
        if (response != null)
    
```

```

        System.out.print("The client received http response, the body is : "
+ new String(response.body()));
        return response;
    }

    public static void main(String[] args) throws Exception {
        HttpClient client = new HttpClient();
        client.connect("127.0.0.1", 18084);

        ByteBuf body = Unpooled.wrappedBuffer("Http message!".getBytes
("UTF-8"));

        DefaultFullHttpRequest request = new DefaultFullHttpRequest
(HttpVersion.HTTP_1_1, HttpMethod.GET,
            "http://127.0.0.1/user?id=10&addr=NanJing", body);

        HttpResponse response = client.blockSend(request);
    }
}

```

进行功能测试，发现 HTTP 服务端处理正常，日志打印如下：

```

HTTP server listening on 18084
Http Server receive the request : HttpObjectAggregator$AggregatedFullHttp-
Request(decodeResult: success, version: HTTP/1.1, content: CompositeByteBuf
(ridx: 0, widx: 13, cap: 13, components=1))
GET http://127.0.0.1/user?id=10&addr=NanJing HTTP/1.1
content-length: 13
Http Server send response succeed : DefaultFullHttpResponse(decodeResult:
success, version: HTTP/1.1, content: PooledUnsafeDirectByteBuf(freed))
HTTP/1.1 200 OK
content-length: 13

```

HTTP 客户端运行异常，如图 4-1 所示。

```

Exception in thread "main" java.lang.UnsupportedOperationException: direct buffer
    at io.netty.buffer.PooledUnsafeDirectByteBuf.array(PooledUnsafeDirectByteBuf.java:343)
    at io.netty.buffer.AbstractUnpooledSlicedByteBuf.array(AbstractUnpooledSlicedByteBuf.java:99)
    at io.netty.buffer.CompositeByteBuf.array(CompositeByteBuf.java:596)
    at io.netty.cases.chapter.demo4.HttpResponse.body(HttpResponse.java:45)
    at io.netty.cases.chapter.demo4.HttpClient.blockSend(HttpClient.java:51)

```

图 4-1 HTTP 客户端获取响应 Body 异常



对 `HttpResponse` 代码进行分析，发现消息体的获取来源是 `Netty FullHttpResponse` 的 `content` 字段，相关代码如下（`HttpResponse` 类）：

---

```
private FullHttpResponse response;

public HttpResponse(FullHttpResponse response)
{
    this.header = response.headers();
    this.response = response;
}

public byte [] body()
{
    return body = response.content() != null ?
        response.content().array() : null;
}
```

---

`response.content().array()`底层调用的是 `PooledUnsafeDirectByteBuf`，查看 `Netty` 源码，发现它并不支持 `array()`方法，相关代码如下（`PooledUnsafeDirectByteBuf` 类）：

---

```
public byte[] array() {
    throw new UnsupportedOperationException("direct buffer");
}
```

---

为了提升性能，`Netty` 默认的 I/O Buffer 使用直接内存 `DirectByteBuf`，可以减少 JVM 用户态到内核态 `Socket` 读写的内存拷贝，即“零拷贝”。由于是直接内存，无法直接转换成堆内存，因此它并不支持 `array()`方法，用户需要自己做内存拷贝操作。

对 `body()`方法进行修改，采用字节拷贝的方式将 HTTP Body 拷贝到 `byte[]`数组中，修改之后的代码如下（`HttpResponse` 类）：

---

```
public byte [] body()
{
    body = new byte[response.content().readableBytes()];
    response.content().getBytes(0, body);
    return body;
}
```

---

修改完成之后做验证，发现客户端再次发生异常，如图 4-2 所示。

```
Exception in thread "main" io.netty.util.IllegalReferenceCountException: refCnt: 0
    at io.netty.buffer.AbstractByteBuf.ensureAccessible(AbstractByteBuf.java:1417)
    at io.netty.buffer.AbstractByteBuf.checkIndex(AbstractByteBuf.java:1356)
    at io.netty.buffer.AbstractByteBuf.checkDstIndex(AbstractByteBuf.java:1376)
    at io.netty.buffer.CompositeByteBuf.getBytes(CompositeByteBuf.java:854)
    at io.netty.buffer.CompositeByteBuf.getBytes(CompositeByteBuf.java:44)
    at io.netty.buffer.AbstractByteBuf.getBytes(AbstractByteBuf.java:474)
    at io.netty.buffer.CompositeByteBuf.getBytes(CompositeByteBuf.java:1740)
    at io.netty.buffer.CompositeByteBuf.getBytes(CompositeByteBuf.java:44)
    at io.netty.cases.chapter.demo4.HttpResponse.body(HttpResponse.java:51)
    at io.netty.cases.chapter.demo4.HttpClient.blockSend(HttpClient.java:51)
```

图 4-2 客户端再次发生异常

### 4.1.2 ByteBuf 非法引用问题

发生 `io.netty.util.IllegalReferenceCountException: refCnt: 0` 异常，说明操作了已经被释放的对象，代码如下（`AbstractByteBuf` 类）：

---

```
protected final void ensureAccessible() {
    if (checkAccessible && refCnt() == 0) {
        throw new IllegalReferenceCountException(0);
    }
}
```

---

`ByteBuf` 实现 `ReferenceCounted` 接口，所以每次操作 `ByteBuf` 之前，都需要对 `ByteBuf` 的生命周期状态进行判断，如果已经被释放，则抛出引用计数异常。根本原因已经定位清楚，但是为何会发生 `IllegalReferenceCountException` 异常呢？

对业务代码进行分析，在收到一个完整的 HTTP 响应消息之后，调用 `respPromise` 的 `setSuccess` 方法，唤醒业务线程继续执行，相关代码如下（`HttpClientHandler` 类）：

---

```
public class HttpClientHandler extends SimpleChannelInboundHandler
<FullHttpResponse> {
    DefaultPromise<FullHttpResponse> respPromise;
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
```

---

```
FullHttpResponse msg) throws Exception {
    if (msg.decoderResult().isFailure())
        throw new Exception("Decode HttpResponse error : " +
msg.decoderResult().cause());
    HttpResponse response = new HttpResponse(msg);
    respPromise.setSuccess(response);
}
}
```

---

客户端调用方的代码如下：

---

```
private HttpResponse blockSend(FullHttpRequest request) throws
InterruptedException, ExecutionException
{
    //代码省略
    DefaultPromise<HttpResponse> respPromise = new DefaultPromise
<HttpResponse>(channel.eventLoop());
    handler.setRespPromise(respPromise);
    channel.writeAndFlush(request);
    HttpResponse response = respPromise.get(); //阻塞调用方线程
    if (response != null)
        System.out.print("The client received http response, the body is :"
+ response.body() );
    return response;
}
```

---

获取 HTTP 响应时抛出非法引用异常，说明 HTTP Body 已经被释放，业务代码并没有主动释放 ByteBuf，ByteBuf 究竟是被谁释放的？

查看 HttpClientHandler 的实现，发现它继承自 SimpleChannelInboundHandler，channelRead0(ChannelHandlerContext ctx, FullHttpResponse msg)方法调用完成之后，Netty 会自动释放 FullHttpResponse，源码如下（SimpleChannelInboundHandler 类）：

---

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
```

---

```

boolean release = true;
try {
    if (acceptInboundMessage(msg)) {
        I imsg = (I) msg;
        channelRead0(ctx, imsg);
    } else {
        release = false;
        ctx.fireChannelRead(msg);
    }
} finally {
    if (autoRelease && release) {
        ReferenceCountUtil.release(msg);
    }
}
}

```

---

由于执行完 `channelRead0` 之后 Netty 的 `NioEventLoop` 线程就会调用 `ReferenceCountUtil.release(msg)` 释放内存，所有后续业务调用方的线程再访问 `FullHttpResponse` 就会出现非法引用问题。

明确问题的根本原因之后，对问题代码进行修改：在 `channelRead0` 中初始化 HTTP Body，此时 `FullHttpResponse` 的 `content` 并没有被释放，可以被访问。修改之后的代码如下（`HttpResponse` 类）：

---

```

public HttpResponse(FullHttpResponse response)
{
    this.header = response.headers();
    this.response = response;
    if (response.content() != null)
    {
        body = new byte[response.content().readableBytes()];
        response.content().getBytes(0, body);
    }
}

```

---

获取 HTTP 响应时，直接返回 body，代码修改如下（HttpResponse 类）：

---

```
public byte [] body()
{
    return body;
}
```

---

修改之后，重新进行验证，HTTP 客户端运行正常，问题解决：

---

The client received http response, the body is :Http message!

---

### 4.1.3 ByteBuf 使用注意事项

---

跨线程操作 ByteBuf，需要重点关注如下两个问题。

（1）ByteBuf 的线程并发安全问题，特别要防止 Netty NioEventLoop 线程与应用线程并发操作 ByteBuf。

（2）ByteBuf 的申请和释放，避免忘记释放、重复释放，以及释放之后继续访问。在这里要重点关注 ByteBuf 的隐式释放问题，例如应用申请了 ByteBuf，但是被 Netty 平台隐式释放了，当应用继续访问或者释放 ByteBuf 时就会发生异常。

此外还要注意性能问题。通常的 get 和 set 方法（类似方法），调用方往往认为获取的是成员变量，不会带来性能问题。但是，如果在此类方法中做复杂的操作，像下面示例中的内存拷贝，就会带来严重的性能问题（每次获取 body 都做一次直接内存到堆内存的拷贝），如果业务频繁访问 body 方法，则会带来严重的性能问题。

---

```
public byte [] body()
{
    if (response.content() == null)
        return null;

    body = new byte[response.content().readableBytes()];
    response.content().getBytes(0, body);
    return body;
}
```

---





## 4.2 Netty ByteBuf 实现机制

Netty 的 ByteBuf 相比原生的 JDK ByteBuffer，功能更丰富，使用起来也更方便，熟悉了它的工作原理之后，就能够在实际项目中灵活运用。

### 4.2.1 Java 原生 ByteBuffer 的局限性

Java NIO 编程使用的主要是 ByteBuffer，但是它有一些局限。

(1) ByteBuffer 只有一个标识位置的指针 position，读写的时候需要手工调用 flip() 和 rewind() 等，使用者必须小心谨慎地处理这些 API，否则很容易导致程序处理失败。

(2) ByteBuffer 长度固定，一旦分配完成，它的容量不能动态扩展和收缩，当需要编码的 POJO 对象大于 ByteBuffer 的容量时，会出现索引越界异常。

(3) ByteBuffer 提供的功能比较少，很多高级功能需要用户自己实现，例如 skipBytes。

### 4.2.2 Netty ByteBuf 工作原理分析

ByteBuf 依然是个 Byte 数组的缓冲区，基本功能应该与 JDK 的 ByteBuffer 一致，它提供以下几类基本功能。

(1) 7 种 Java 基础类型、byte 数组、ByteBuffer (ByteBuf) 等的读写。

(2) 缓冲区自身的 copy 和 slice 等。

(3) 操作位置指针等方法。

(4) 容量自动扩展。

(5) 从 ByteBuf 到其他数据结构的灵活转换（例如 JDK ByteBuffer）。

不同于 JDK ByteBuffer 的 limit，ByteBuf 通过两个位置指针来协助缓冲区的读写操作，读操作使用 readerIndex，写操作使用 writerIndex。



`readerIndex` 和 `writerIndex` 的取值开始都是 0，随着数据的写入 `writerIndex` 会增加，读取数据会使 `readerIndex` 增加，但是它不会超过 `writerIndex`。在读取之后，`0~readerIndex` 的数据就被视为被丢弃的（discard），调用 `discardReadBytes` 方法，可以释放这部分空间，它的作用类似于 `ByteBuffer` 的 `compact` 方法。`ReaderIndex` 和 `writerIndex` 之间的数据是可读取的，等价于 `ByteBuffer` `position` 和 `limit` 之间的数据。`WriterIndex` 和 `capacity` 之间的空间是可写的，等价于 `ByteBuffer` `limit` 和 `capacity` 之间的可用空间。

由于写操作不修改 `readerIndex` 指针，读操作不修改 `writerIndex` 指针，因此读写之间不再需要调整位置指针，这极大地简化了缓冲区的读写操作，避免了由于遗漏或者不熟悉 `JDK ByteBuffer` 的 `flip()` 操作导致功能异常。

Netty 提供了两个指针变量用于支持顺序读取和写入操作：`readerIndex` 用于标识读取索引，`writerIndex` 用于标识写入索引。两个位置指针将 `ByteBuf` 缓冲区划分为三部分，如图 4-3 所示。

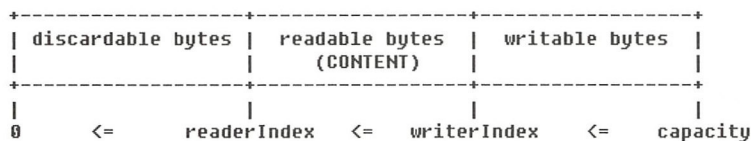


图 4-3 两个位置指针将 `ByteBuf` 缓冲区划分为三部分

调用 `ByteBuf` 的 `read` 操作时，从 `readerIndex` 处开始读取。从 `readerIndex` 到 `writerIndex` 的空间为可读的字节缓冲区；从 `writerIndex` 到 `capacity` 的空间为可写的字节缓冲区；从 0 到 `readerIndex` 的空间是已经读取的缓冲区，可以调用 `discardReadBytes` 操作来重用这部分空间，以节约内存，防止 `ByteBuf` 不断扩张。

刚创建的 `ByteBuf` 位置指针如图 4-4 所示。

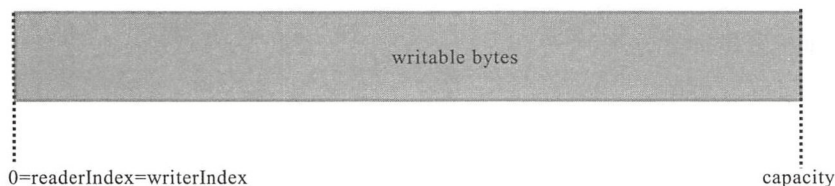
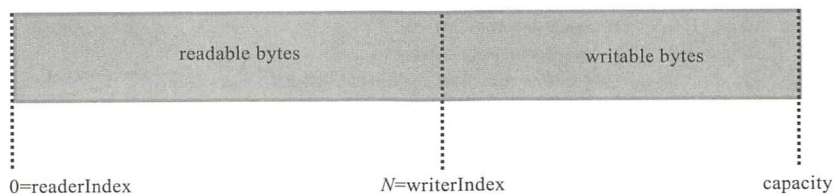


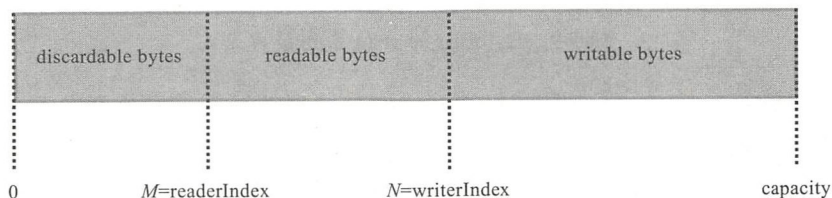
图 4-4 刚创建的 `ByteBuf` 位置指针

写入  $N$  个字节之后 `ByteBuf` 的位置指针如图 4-5 所示。



图 4-5 写入  $N$  个字节之后 ByteBuf 的位置指针

读取  $M$  ( $<N$ ) 个字节后 ByteBuf 的指针如图 4-6 所示。

图 4-6 读取  $M$  个字节后 ByteBuf 的指针

调用 discard 操作后 ByteBuf 的指针如图 4-7 所示。

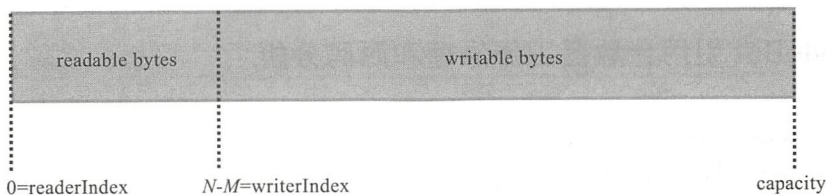


图 4-7 调用 discard 操作后 ByteBuf 的指针

调用 clear 操作后, readerIndex 和 writerIndex 被重置为 0, 如图 4-8 所示。

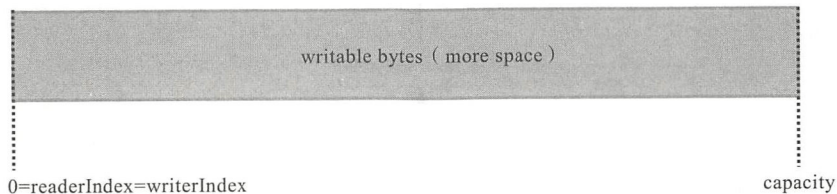


图 4-8 调用 clear 操作后 ByteBuf 的指针

ByteBuf 支持顺序和随机读写, 其中 read 对应于 JDK ByteBuffer 的 get 操作, write 对应于 put 操作, ByteBuf 的 read 系列接口定义如图 4-9 所示。



```

m  ↳ readByte(): byte ↳ ByteBuf
m  ↳ readBytes(byte[]): ByteBuf ↳ ByteBuf
m  ↳ readBytes(byte[], int, int): ByteBuf ↳ ByteBuf
m  ↳ readBytes(ByteBuf): ByteBuf ↳ ByteBuf
m  ↳ readBytes(ByteBuf, int): ByteBuf ↳ ByteBuf
m  ↳ readBytes(ByteBuf, int, int): ByteBuf ↳ ByteBuf
m  ↳ readBytes(ByteBuffer): ByteBuf ↳ ByteBuf
m  ↳ readBytes(FileChannel, long, int): int ↳ ByteBuf
m  ↳ readBytes(GatheringByteChannel, int): int ↳ ByteBuf
m  ↳ readBytes(int): ByteBuf ↳ ByteBuf
m  ↳ readBytes(OutputStream, int): ByteBuf ↳ ByteBuf
m  ↳ readChar(): char ↳ ByteBuf
    
```

图 4-9 ByteBuf 的 read 系列接口定义

除了顺序读写，Netty 还支持随机读写，它与顺序读写的最大差别在于，可以随机指定读写的索引位置。无论 get 还是 set 操作，ByteBuf 都会对其索引和长度等进行合法性校验，与顺序读写一致。但是，set 操作与 write 操作不同的是，它不支持动态扩展缓冲区，所以使用者必须保证当前的缓冲区可写的字节数大于需要写入的字节数，否则会抛出数组或者缓冲区越界异常。

### 4.2.3 ByteBuf 引用计数器工作原理和源码分析

AbstractReferenceCountedByteBuf 实现了对 ByteBuf 的内存管理，通过引用计数器对 ByteBuf 的引用情况进行管理，跟踪对象的分配和使用情况，以实现内存的回收、销毁和重用。它的继承关系如图 4-10 所示。

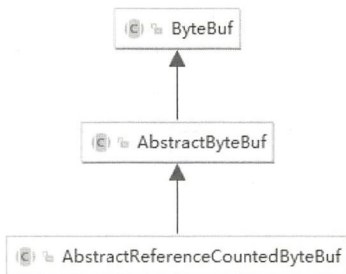


图 4-10 AbstractReferenceCountedByteBuf 的继承关系

AbstractReferenceCountedByteBuf 的 refCnt 是引用计数器，用于对 ByteBuf 的申请和释放做引用计数。它的更新通过原子类 AtomicIntegerFieldUpdater 完成。AtomicIntegerFieldUpdater



通过原子的方式对成员变量进行更新等操作，以实现线程安全，避免加锁，提升性能，示例代码如下（AbstractReferenceCountedByteBuf）：

---

```
public abstract class AbstractReferenceCountedByteBuf extends AbstractByteBuf {
    private static final AtomicIntegerFieldUpdater<AbstractReferenceCounted-
ByteBuf> refCntUpdater =
        AtomicIntegerFieldUpdater.newUpdater(AbstractReferenceCounted-
ByteBuf.class, "refCnt");
    private volatile int refCnt;
    protected AbstractReferenceCountedByteBuf(int maxCapacity) {
        super(maxCapacity);
        refCntUpdater.set(this, 1);
    }
}
//后续代码省略
```

---

refCnt 使用 volatile 修饰解决多线程并发访问时的线程可见性问题，即如果有一个线程修改了 refCnt 的值，其他线程将立即读取到最新的 refCnt，而不是之前的脏数据。

每调用一次 retain 方法，引用计数器就会累加一次，由于可能存在多线程并发调用的场景，所以它的累加操作必须是线程安全的，通过自旋对引用计数器进行累加操作，如果执行累加操作后发现引用计数器小于等于 0，或者发生了整型范围越界（例如  $0x7fffffff + 1$ ），则抛出 IllegalReferenceCountException 异常，代码如下（AbstractReferenceCountedByteBuf 类）：

---

```
private ByteBuf retain0(final int increment) {
    int oldRef = refCntUpdater.getAndAdd(this, increment);
    if (oldRef <= 0 || oldRef + increment < oldRef) {
        refCntUpdater.getAndAdd(this, -increment);
        throw new IllegalReferenceCountException(oldRef, increment);
    }
    return this;
}
```

---

通过调用 release 方法释放引用，如果 oldRef 与 decrement 相等，说明对象申请和释放次数相同，refCnt 为 0，ByteBuf 已经不再被引用，需要执行释放操作。如果发生了重复释





放 ( $\text{refCnt} < 0$ ) 和越界操作, 则抛出非法引用异常, 代码如下 (`AbstractReferenceCounted-ByteBuf` 类):

---

```
private boolean release0(int decrement) {
    int oldRef = refCntUpdater.getAndAdd(this, -decrement);
    if (oldRef == decrement) {
        deallocate();
        return true;
    } else if (oldRef < decrement || oldRef - decrement > oldRef) {
        refCntUpdater.getAndAdd(this, decrement);
        throw new IllegalReferenceCountException(oldRef, -decrement);
    }
    return false;
}
```

---

需要注意的是, 不同的 `ByteBuf` 实现, 释放策略不同, 如果是内存池模式, 则需要将 `ByteBuf` 返回到内存池中以便重用, `PooledByteBuf` 的 `deallocate` 代码实现如下:

---

```
protected final void deallocate() {
    if (handle >= 0) {
        final long handle = this.handle;
        this.handle = -1;
        memory = null;
        tmpNioBuf = null;
        chunk.arena.free(chunk, handle, maxLength, cache);
        chunk = null;
        recycle();
    }
}
```

---

通过 `chunk.arena.free` 和 `recycle` 调用, 释放并重用 `ByteBuf`。对于使用非池模式创建的 `ByteBuf`, 则释放 `ByteBuf` 相关资源 (设置为 `null`), 等待 GC 回收, 以 `UnpooledDirectByteBuf` 为例, 代码如下:

---

```
protected void deallocate() {
```

---



```
ByteBuffer buffer = this.buffer;
if (buffer == null) {
    return;
}
this.buffer = null;
if (!doNotFree) {
    freeDirect(buffer);
}
}
```

---

## 4.3 总结

ByteBuf 的申请和释放可能会跨 Netty 的 `NioEventLoop` 和业务线程，跨线程操作 ByteBuf 时一定要谨慎，防止发生并发安全和非法引用问题。另外，由于 ByteBuf 的实现类非常多，不同的实现功能特性存在差异，用户在使用时一定要认真阅读 API Doc 说明，必要时要看源码，防止误用导致出现功能和性能问题。



## 第 5 章

---

# Netty 发送队列积压导致内存泄漏案例

导致 Netty 内存泄漏的原因很多，例如，使用内存池方式创建的对象忘记释放，或者系统处理压力过大导致发送队列积压。

尽管 Netty 采用了 NIO 非阻塞通信，I/O 处理往往不会成为业务瓶颈，但是如果客户端并发压力过大，超过了服务端的处理能力，又没有流控保护，则很容易发生内存泄漏。

## 5.1 Netty 发送队列积压案例

---

对某业务进行性能压测， $N$  个客户端并发访问服务端，客户端基于 Netty 框架做网络通信，压测一段时间之后，响应时间越来越长，失败率增加，监控客户端内存使用情况，发现使用的内存一直飙升，最后发生 OOM 异常，CPU 占用率居高不下，吞吐量降为 0。

### 5.1.1 高并发故障场景

---

为了便于分析，对真实的业务代码做简化处理，在一个客户端内部创建一个线程，向

服务端循环发送请求消息，模拟客户端高并发场景，示例代码如下：

```
public void channelActive(final ChannelHandlerContext ctx) {  
    loadRunner = new Runnable() {  
        @Override  
        public void run() {  
            try {  
                TimeUnit.SECONDS.sleep(30);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            ByteBuf msg = null;  
            final int len = "Netty OOM Example".getBytes().length;  
            while(true)  
            {  
                msg = Unpooled.wrappedBuffer("Netty OOM Example".getBytes());  
                ctx.writeAndFlush(msg);  
            }  
        }  
    };  
    new Thread(loadRunner, "LoadRunner-Thread").start();  
}
```

创建链路之后，客户端启动一个线程向服务端循环发送请求消息，模拟客户端压测场景，系统运行一段时间之后，发现内存占用率飙升，内存数据监控如图 5-1 所示。

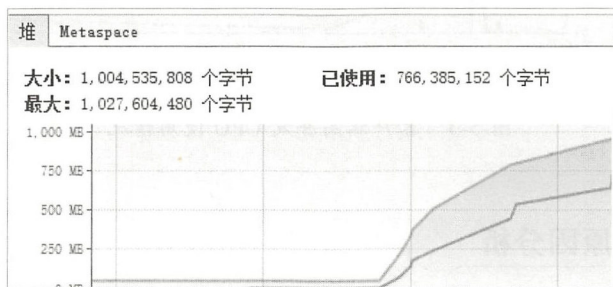


图 5-1 客户端高并发场景内存数据监控



统计 GC 数据，发现老年代已满，发生多次 Full GC，耗时 3 分钟多，系统已经无法正常运行，如图 5-2 所示。

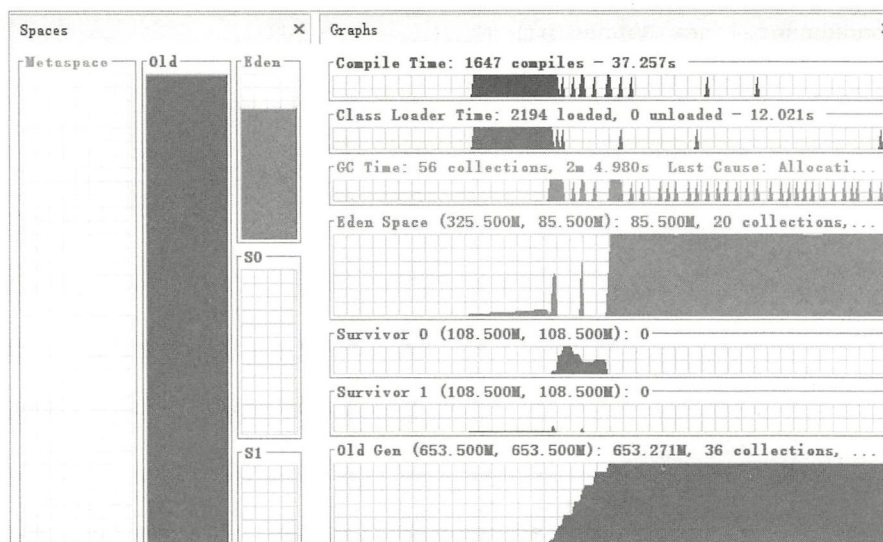


图 5-2 客户端高并发 GC 数据采集

查看 CPU 的使用情况，发现 GC 线程占用了大量的 CPU 资源，如图 5-3 所示。

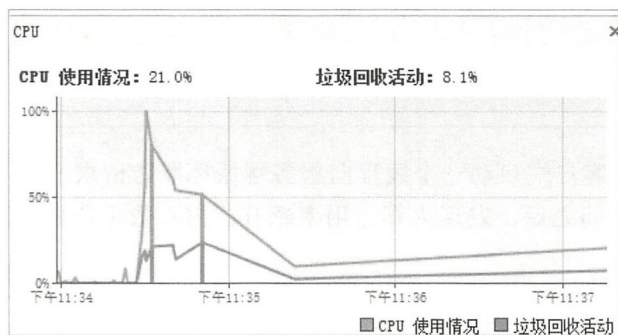


图 5-3 客户端高并发 CPU 使用情况

### 5.1.2 内存泄漏原因分析

Dump 客户端内存文件进行分析，发现 Netty 的 NioEventLoop 占用了 99.7%内存，可



以确认 NioEventLoop 发生了内存泄漏，如图 5-4 所示。

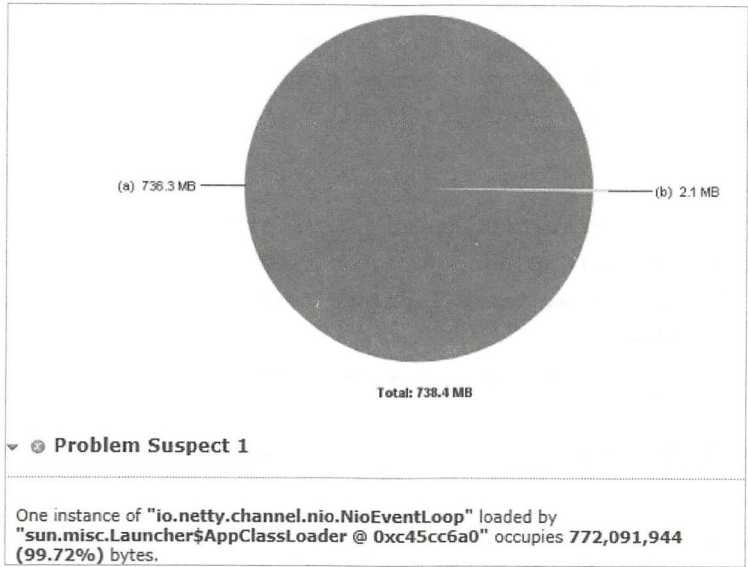


图 5-4 内存泄漏视图分析

继续对引用关系进行分析，发现真正泄漏的对象是 WriteAndFlushTask，它包含了待发送的客户端请求消息 msg 及 promise 对象，引用关系示例如图 5-5 所示。

io.netty.util.internal.shaded.org.jctools.queues.MpscUnboundedArrayQueue @ 0xc4698a98	560	772,086,712
> <class> class io.netty.util.internal.shaded.org.jctools.queues.MpscUnboundedArrayQueue @ 0xc4698a98	0	0
> consumerBuffer java.lang.Object[1025] @ 0xc469c4b0	4,120	771,966,832
> <class> class java.lang.Object[] @ 0xc45ed6c8	0	0
> [1023] io.netty.channel.AbstractChannelHandlerContext\$WriteAndFlushTask @ 0xc2dd9178	32	192
> <class> class io.netty.channel.AbstractChannelHandlerContext\$WriteAndFlushTask @ 0xc2dd9178	8	8
> handle io.netty.util.Recycler\$DefaultHandle @ 0xc2dd9198	32	32
> msg io.netty.buffer.UnpooledHeapByteBuf @ 0xc2dd91b8	48	88
> <class> class io.netty.buffer.UnpooledHeapByteBuf @ 0xc2e64a48	0	0
> array byte[17] @ 0xc2dd91e8 Netty OOM Example	40	40
> alloc io.netty.buffer.UnpooledByteBufAllocator @ 0xc45edc08	32	120
Σ Total: 3 entries		
> promise io.netty.channel.DefaultChannelPromise @ 0xc2dd9210	40	40
> ctx io.netty.channel.DefaultChannelPipeline\$HeadContext @ 0xc45ee248	72	72
Σ Total: 5 entries		

图 5-5 内存泄漏对象引用关系

Netty 的消息发送队列为什么会积压呢？通过源码分析发现，调用 Channel 的 write 方法时，如果发送方为业务线程，则将发送操作封装成 WriteTask，放到 Netty 的 NioEventLoop 中执行，源码如下（AbstractChannelHandlerContext）：



```
private void write(Object msg, boolean flush, ChannelPromise promise) {
    AbstractChannelHandlerContext next = findContextOutbound();
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        //代码省略
    } else {
        AbstractWriteTask task;
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        } else {
            task = WriteTask.newInstance(next, m, promise);
        }
        safeExecute(executor, task, promise, m);
    }
}
```

显然 Netty 的 I/O 线程 `NioEventLoop` 无法完成如此多消息的发送，因此发送任务队列积压，进而导致内存泄漏。

### 5.1.3 如何防止发送队列积压

为了防止在高并发场景下，由于服务端处理慢导致客户端消息积压，除了服务端做流控，客户端也需要做并发保护，防止自身发生消息积压。

利用 Netty 提供的高低水位机制，可以实现客户端更精准的流控，它的高水位接口说明如图 5-6 所示。

```
/**
 * <p>
 * Sets the high water mark of the write buffer. If the number of bytes
 * queued in the write buffer exceeds this value, {@link Channel#isWritable\(\)}
 * will start to return {@code false}.
 * </p>
 */
ChannelConfig setWriteBufferHighWaterMark(int writeBufferHighWaterMark);
```

图 5-6 Netty 高水位接口说明



当发送队列待发送的字节数组达到高水位时，对应的 Channel 就变为不可写状态。由于高水位并不影响业务线程调用 write 方法并把消息加入待发送队列，因此，必须在消息发送时对 Channel 的状态进行判断：当到达高水位时，Channel 的状态被设置为不可写，通过对 Channel 的可写状态进行判断来决定是否发送消息。

按照上述理念对问题代码进行修改，修改之后的代码如下：

---

```
public void channelActive(final ChannelHandlerContext ctx) {
    ctx.channel().config().setWriteBufferHighWaterMark(10 * 1024 * 1024);
    loadRunner = new Runnable() {
        @Override
        public void run() {
            try {
                TimeUnit.SECONDS.sleep(30);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            ByteBuf msg = null;
            while (true) {
                if (ctx.channel().isWritable()) {
                    msg = Unpooled.wrappedBuffer("Netty OOM Example".getBytes());
                    ctx.writeAndFlush(msg);
                } else {
                    LOG.warning("The write queue is busy : " +
ctx.channel().unsafe().outboundBuffer().nioBufferSize());
                }
            }
        }
    };
    new Thread(loadRunner, "LoadRunner-Thread").start();
}
```

---

修改之后进行验证，客户端代码中打印队列积压相关日志，说明基于高水位的流控机制生效，日志如下：



---

警告：The write queue is busy : 17

---

通过内存和 CPU 使用情况监控，发现指标正常，内存泄漏问题解决，如图 5-7 和图 5-8 所示。

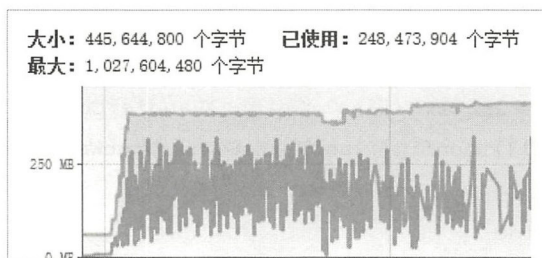


图 5-7 优化之后内存使用情况

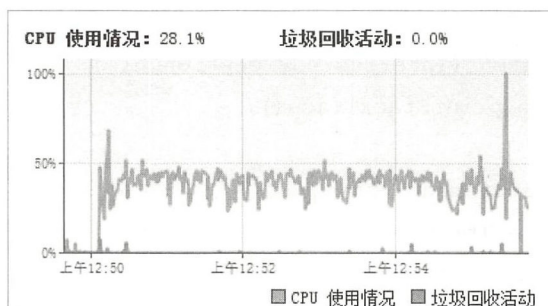


图 5-8 优化之后 CPU 使用情况

在实际项目中，根据业务 QPS 规划、客户端处理性能、网络带宽、链路数、消息平均码流大小等综合因素计算并设置高水位（WriteBufferHighWaterMark）值，利用高水位做消息发送速度的流控，既可以保护自身，同时又能减轻服务端的压力，防止服务端被压挂。

#### 5.1.4 其他可能导致发送队列积压的因素

需要指出的是，并非只有高并发场景才会导致消息积压，在一些异常场景下，尽管系统流量不大，但仍然可能导致消息积压，可能的场景如下。

（1）网络瓶颈，当发送速度超过网络链接处理能力，会导致发送队列积压。



(2) 当对端读取速度小于己方发送速度, 导致自身 TCP 发送缓冲区满, 频繁发生 write 0 字节时, 待发送消息会在 Netty 发送队列中排队。

当出现大量排队时, 很容易导致 Netty 的直接内存泄漏。对案例中的代码做改造, 模拟直接内存泄漏, 思路如下: 服务端在消息接收处 Debug, 模拟服务端处理慢, 不读取网络消息; 客户端每 1ms 发送一条消息, 由于服务端不读取网络消息, 会导致客户端的发送队列积压。

客户端代码改造如下 (LoadRunnerSleepClientHandler):

---

```
public void channelActive(final ChannelHandlerContext ctx) {
    loadRunner = new Runnable() {
        @Override
        public void run() {
            ByteBuf msg = null;
            while(true)
            {
                byte [] body = new byte[SIZE];
                msg = Unpooled.wrappedBuffer(body);
                ctx.writeAndFlush(msg);
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    };
    new Thread(loadRunner, "LoadRunner-Thread").start();
}
```

---

服务端在 channelRead 中设置断点, 模拟阻塞 NioEventLoop 线程, 因为 Netty 在发送消息时会把堆内存转换成直接内存, 通过内存监控无法直观地看到直接内存的分配和使用情况, 所以运行一段时间之后可以在客户端 AbstractUnsafe 的 write 处设置断点, 查看发送队列 (ChannelOutboundBuffer) 堆积情况, 断点如图 5-9 所示。





## Netty 进阶之路：跟着案例学 Netty

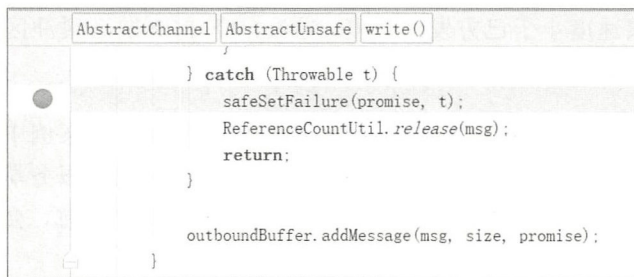


图 5-9 消息发送内存泄漏断点调试

异常堆栈如图 5-10 所示，发生 OOM 异常，Netty 直接内存分配失败。

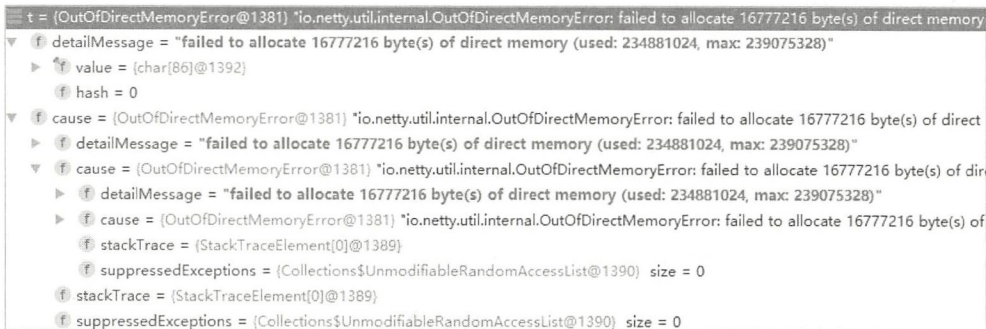


图 5-10 发送队列积压导致 OOM 异常堆栈

利用 `netstat -ano` 等命令可以监控某个端口的 TCP 接收（recv-q）和发送（send-q）队列的积压情况，一旦发现己方的发送队列有大量积压，说明消息的收发存在瓶颈，需要及时解决，防止因 Netty 发送队列积压而导致内存泄漏。在日常监控中，需要将 Netty 的链路数、网络读写速度等指标纳入监控系统，发现问题之后需要及时告警。

## 5.2 Netty 消息发送工作机制

业务调用 `write` 方法后，经过 `ChannelPipeline` 职责链处理，消息被投递到发送缓冲区待发送，调用 `flush` 之后会执行真正的发送操作，底层通过调用 Java NIO 的 `SocketChannel` 进行非阻塞 `write` 操作，将消息发送到网络上，它的工作原理如图 5-11 所示。

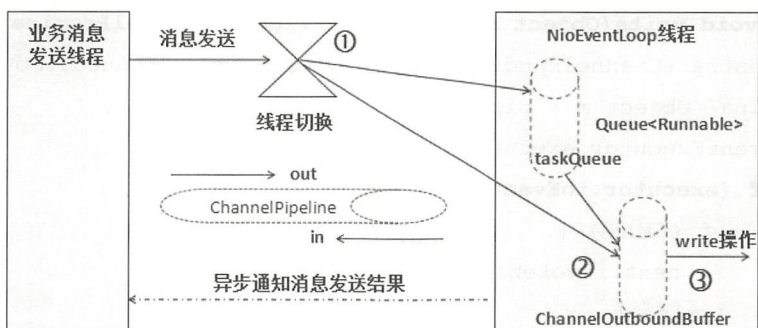


图 5-11 Netty 消息发送工作原理

Netty 的消息发送涉及线程切换、消息队列、高低水位和写半包消息，实现比较复杂，下面我们对主要的技术点进行分析。

### 5.2.1 WriteAndFlushTask 原理和源码分析

为了尽可能地提升性能，Netty 采用了串行无锁化设计，在 I/O 线程内部进行串行操作，避免多线程竞争导致性能下降。从表面看，串行化设计的 CPU 利用率似乎不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比“一个队列对应多个工作线程”模型性能更优。

Netty 的串行化工作原理如图 5-12 所示。

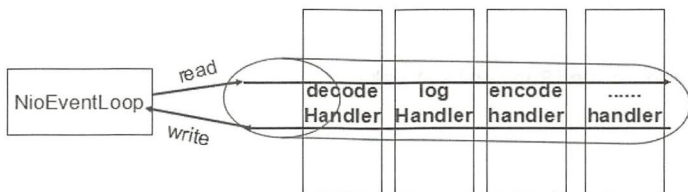


图 5-12 Netty 的串行化工作原理

当用户线程发起 write 操作时，Netty 会进行判断，如果发现不是 NioEventLoop (I/O 线程)，则将发送消息封装成 WriteTask，放入 NioEventLoop 的任务队列由 NioEventLoop 线程执行，代码如下 (AbstractChannelHandlerContext 类)：



---

```
private void write(Object msg, boolean flush, ChannelPromise promise) {
    AbstractChannelHandlerContext next = findContextOutbound();
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            next.invokeWrite(m, promise);
        }
    } else {
        AbstractWriteTask task;
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        } else {
            task = WriteTask.newInstance(next, m, promise);
        }
        safeExecute(executor, task, promise, m);
    }
}
```

---

Netty 的 `NioEventLoop` 线程内部维护了一个 `Queue<Runnable> taskQueue`, 除了处理网络 I/O 读写操作, 同时还负责执行网络读写相关的 Task (包含用户自定义 Task), 代码如下 (`SingleThreadEventExecutor` 类):

---

```
public void execute(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }
    boolean inEventLoop = inEventLoop();
    addTask(task);
    if (!inEventLoop) {
        startThread();
        if (isShutdown() && removeTask(task)) {
```

---

```

        reject();
    }
}

if (!addTaskWakesUp && wakesUpForTask(task)) {
    wakeup(inEventLoop);
}
}
}

```

---

NioEventLoop 遍历 taskQueue, 执行消息发送任务, 代码如下 (AbstractWriteTask 类):

---

```

public final void run() {
    try {
        if (ESTIMATE_TASK_SIZE_ON_SUBMIT) {
            ctx.pipeline.decrementPendingOutboundBytes(size);
        }
        write(ctx, msg, promise);
    } finally {
        //代码省略
    }
}
}

```

---

经过一些系统处理操作, 最终会调用 ChannelOutboundBuffer 的 addMessage 方法, 将发送消息加入发送队列 (数据结构为链表)。

通过上述分析, 可以得出几点结论。

(1) 多个业务线程并发调用 write 相关方法是线程安全的, Netty 会将发送消息封装成 Task, 由 I/O 线程异步执行。

(2) 由于单个 Channel 由其对应的 NioEventLoop 线程 (单个 NioEventLoop) 执行, 如果并行调用某个 Channel 的 write 操作超过对应的 NioEventLoop 线程的执行能力, 则会导致 WriteTask 积压。

(3) NioEventLoop 线程需要处理网络读写操作, 以及注册到 NioEventLoop 上的各种 Task, 两者相互影响, 如果网络读写任务较重, 或者注册的 Task 过多, 都会导致对方延迟执行, 引发性能问题。

## 5.2.2 ChannelOutboundBuffer 原理和源码分析

ChannelOutboundBuffer 是 Netty 的发送缓冲队列，它基于链表来管理待发送的消息，定义如下（ChannelOutboundBuffer 类）：

---

```

    private Entry flushedEntry;
    private Entry unflushedEntry;
    private Entry tailEntry;
    private int flushed;
}
//Entry 定义如下
static final class Entry {
    //代码省略
    private final Handle<Entry> handle;
    Entry next;
    Object msg;
    ByteBuffer[] bufs;
    ByteBuffer buf;
    ChannelPromise promise;
}
//后续代码省略

```

---

在消息发送时会调用 ChannelOutboundBuffer 的 addMessage 方法，修改链表指针，将新加入的消息放到尾部，同时更新上一个尾部消息的 next 指针，指向新加入的消息，代码如下：

---

```

public void addMessage(Object msg, int size, ChannelPromise promise) {
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);
    if (tailEntry == null) {
        flushedEntry = null;
    } else {
        Entry tail = tailEntry;
        tail.next = entry;
    }
    tailEntry = entry;
}

```

---



```

    if (unflushedEntry == null) {
        unflushedEntry = entry;
    }
    //后续代码省略

```

---

在消息发送时，调用 `current` 方法，获取待发送的原始消息（`flushedEntry`）：

---

```

public Object current() {
    Entry entry = flushedEntry;
    if (entry == null) {
        return null;
    }
    return entry.msg;
}
//后续代码省略

```

---

如果消息发送成功，则调用 `ChannelOutboundBuffer` 的 `remove` 方法，将已发送消息从链表中删除，同时更新待发送的消息，代码如下：

---

```

private void removeEntry(Entry e) {
    if (-- flushed == 0) {
        flushedEntry = null;
        if (e == tailEntry) {
            tailEntry = null;
            unflushedEntry = null;
        }
    } else {
        flushedEntry = e.next;
    }
}
//后续代码省略

```

---

如果队列中已无剩余待发送消息，则把 `flushedEntry` 设置为 `null`，否则将 `flushedEntry` 设置为链表中下一个待发送的消息。

将已发送的消息从链表中删除后，释放 `ByteBuf` 资源，如果是基于内存池分配的

ByteBuf, 则重新返回池中重用; 如果是非池模式, 则清空相关资源, 等待 GC 回收, 代码如下 (ChannelOutboundBuffer):

```
public boolean remove() {
    //代码省略
    if (!e.cancelled) {
        ReferenceCountUtil.safeRelease(msg);
        safeSuccess(promise);
        //代码省略
    }
}
```

### 5.2.3 消息发送源码分析

Netty 消息发送原理如图 5-13 所示。

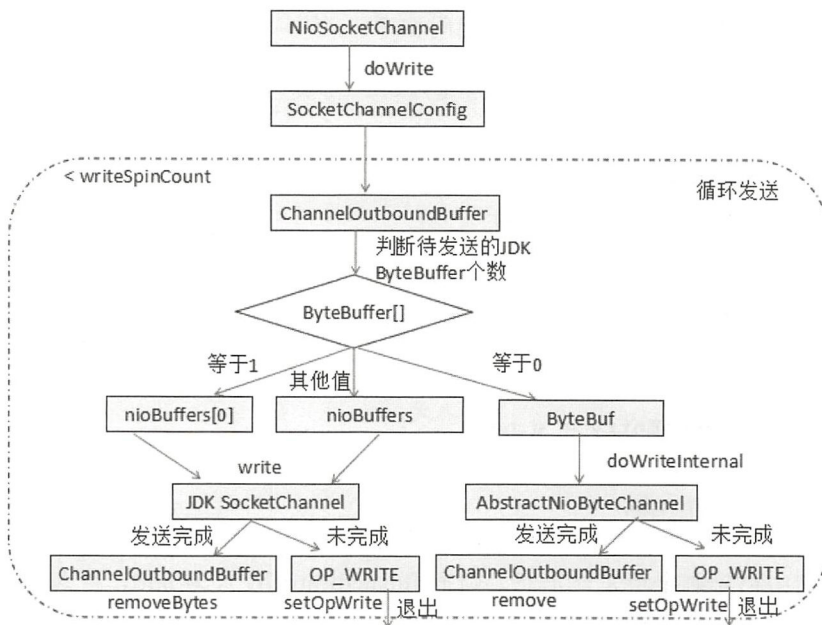


图 5-13 Netty 消息发送原理

## 1. 发送次数限制

当 `SocketChannel` 无法一次将所有待发送的 `ByteBuf/ByteBuffer` 写入网络时, 需要决定是注册 `SelectionKey.OP_WRITE` 在下一次 `Selector` 轮询时继续发送, 还是在当前位置循环发送, 等到所有消息都发送完成再返回。

如果频繁地注册 `SelectionKey.OP_WRITE` 并 wakeup `Selector` 会影响性能; 但是如果 TCP 的发送缓冲区已满, TCP 处于 `KEEP-ALIVE` 状态, 消息无法发送出去, 如果不对循环发送次数进行控制, 就会长时间处于发送状态, `Reactor` 线程无法及时读取其他消息和执行排队的 `Task`。所以 Netty 采取了折中的方式, 即如果本次发送的字节数大于 0, 但是消息尚未发送完, 则循环发送, 一旦发现 `write` 字节数为 0, 说明 TCP 缓冲区已满, 此时继续发送没有意义, 注册 `SelectionKey.OP_WRITE` 并退出循环, 在下一个 `SelectionKey` 轮询周期继续发送, 代码如下 (`NioSocketChannel` 类):

---

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    SocketChannel ch = javaChannel();
    int writeSpinCount = config().getWriteSpinCount();
    do {
        //消息发送代码, 此处省略
    } while (writeSpinCount > 0);
    incompleteWrite(writeSpinCount < 0);
}
```

---

## 2. 不同的消息发送策略

消息发送有三种策略。

(1) 如果待发送消息 (`ChannelOutboundBuffer`) 的 `ByteBuffer` 数等于 1, 则直接通过 `nioBuffers[0]` 获取待发送消息的 `ByteBuffer`, 通过调用 JDK 的 `SocketChannel` 直接完成消息发送, 代码如下 (`NioSocketChannel` 类 `doWrite` 方法):

---

```
for (;;) {
    ByteBuffer[] nioBuffers = in.nioBuffers(1024,
maxBytesPerGatheringWrite);
    int nioBufferCnt = in.nioBufferCount();
    switch (nioBufferCnt) {
```

---

```

//代码省略
case 1: {
    ByteBuffer buffer = nioBuffers[0];
    int attemptedBytes = buffer.remaining();
    final int localWrittenBytes = ch.write(buffer);
    if (localWrittenBytes <= 0) {
        incompleteWrite(true);
        return;
    }
    adjustMaxBytesPerGatheringWrite(attemptedBytes,
localWrittenBytes, maxBytesPerGatheringWrite);
    in.removeBytes(localWrittenBytes);
    --writeSpinCount;
    break;
}
}

```

---

(2) 如果待发送消息的 ByteBuffer 数大于 1，则调用 SocketChannel 的批量发送接口，将 nioBuffers 数组写入 TCP 发送缓冲区，代码如下：

```

default: {
    long attemptedBytes = in.nioBufferSize();
    final long localWrittenBytes = ch.write(nioBuffers, 0,
nioBufferCnt);
    if (localWrittenBytes <= 0) {
        incompleteWrite(true);
        return;
    }
    adjustMaxBytesPerGatheringWrite((int) attemptedBytes,
(int) localWrittenBytes,
        maxBytesPerGatheringWrite);
    in.removeBytes(localWrittenBytes);
    --writeSpinCount;
    break;
}
}

```

---

(3) 如果待发送的消息包含的 JDK 原生 `ByteBuffer` 数为 0, 则调用父类 `AbstractNioByteChannel` 的 `doWrite0` 方法, 将 Netty 的 `ByteBuf` 发到 TCP 缓冲区, 代码如下 (`AbstractNioByteChannel` 类):

---

```
private int doWriteInternal(ChannelOutboundBuffer in, Object msg) throws
Exception {
    if (msg instanceof ByteBuf) {
        ByteBuf buf = (ByteBuf) msg;
        if (!buf.isReadable()) {
            in.remove();
            return 0;
        }
        final int localFlushedAmount = doWriteBytes(buf);
        if (localFlushedAmount > 0) {
            in.progress(localFlushedAmount);
            if (!buf.isReadable()) {
                in.remove();
            }
            return 1;
        }
    }
    //后续代码省略
}
```

---

### 3. 已发送消息内存释放

如果消息被发送成功, Netty 会释放已发送消息的内存, 发送的对象不同, 释放策略也不同。

(1) 如果发送对象是 JDK 的 `ByteBuffer`, 则根据发送的字节数计算需要被释放的发送对象个数, 代码如下 (`ChannelOutboundBuffer` 类):

---

```
public void removeBytes(long writtenBytes) {
    for (;;) {
        Object msg = current();
        if (!(msg instanceof ByteBuf)) {
            assert writtenBytes == 0;
        }
    }
}
```

---



```

        break;
    }
    final ByteBuf buf = (ByteBuf) msg;
    final int readerIndex = buf.readerIndex();
    final int readableBytes = buf.writerIndex() - readerIndex;
    if (readableBytes <= writtenBytes) {
        if (writtenBytes != 0) {
            progress(readableBytes);
            writtenBytes -= readableBytes;
        }
        remove();
    } else {
        if (writtenBytes != 0) {
            buf.readerIndex(readerIndex + (int) writtenBytes);
            progress(writtenBytes);
        }
        break;
    }
}

clearNioBuffers();
}

```

---

对可读字节数和发送的总字节数进行比较，如果发送的字节数大于可读的字节数，说明当前的 `ByteBuffer` 已经被完全发送出去，更新 `ChannelOutboundBuffer` 的发送进度信息，将已经发送的 `ByteBuffer` 删除，释放相关资源。最后，发送的字节数要减去第一条消息的字节数，得到后续消息发送的总字节数，然后继续循环判断第二条消息、第三条消息……直到所有已发送消息都被删除。

(2) 如果发送对象是 Netty 的 `ByteBuf`，则通过判断当前 `ByteBuf` 的 `isReadable` 来获取消息发送结果，如果发送完成，则调用 `ChannelOutboundBuffer` 的 `remove` 方法删除并释放 `ByteBuf`，代码如下（`AbstractNioByteChannel` 类 `doWriteInternal` 方法）：

---

```

//代码省略
final int localFlushedAmount = doWriteBytes(buf);

```

```

        if (localFlushedAmount > 0) {
            in.progress(localFlushedAmount);
            if (!buf.isReadable()) {
                in.remove();
            }
            return 1;
        }
    }
}

```

---

#### 4. 写半包

如果一次无法将待发送的消息全部写入 TCP 缓冲区, 循环 writeSpinCount 次仍然未发送完, 或者在发送过程中出现了 TCP 零滑窗 (写入的字节数为 0), 则进入“写半包”模式 (目的是在消息发送慢时不要死循环发送, 这会阻塞 NioEventLoop 线程), 注册 SelectionKey.OP\_WRITE 到对应的 Selector, 退出循环, 在下次 Selector 轮询过程中继续执行 write 操作, 代码如下 (NioSocketChannel 类 doWrite 方法):

```

        if (localWrittenBytes <= 0) {
            incompleteWrite(true);
            return;
        }
    }
}

```

---

注册 SelectionKey.OP\_WRITE 相关代码如下 (AbstractNioByteChannel 类 setOpWrite 方法):

```

protected final void setOpWrite() {
    final SelectionKey key = selectionKey();
    if (!key.isValid()) {
        return;
    }
    final int interestOps = key.interestOps();
    if ((interestOps & SelectionKey.OP_WRITE) == 0) {
        key.interestOps(interestOps | SelectionKey.OP_WRITE);
    }
}

```

---

## 5.2.4 消息发送高低水位控制

为了对发送速度和消息积压数进行控制，Netty 提供了高低水位机制，当消息队列中积压的待发送消息总字节数到达高水位时，修改 Channel 的状态为不可写，代码如下（ChannelOutboundBuffer 类）：

---

```
private void incrementPendingOutboundBytes(long size, boolean invokeLater) {
    if (size == 0) {
        return;
    }
    long newWriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this, size);
    if (newWriteBufferSize > channel.config().getWriteBufferHighWater
Mark()) {
        setUnwritable(invokeLater);
    }
}
}
```

---

修改 Channel 状态后，调用 ChannelPipeline 发送通知事件，业务可以监听该事件及时获取链路可写状态，代码如下（ChannelOutboundBuffer 类）：

---

```
private void fireChannelWritabilityChanged(boolean invokeLater) {
    final ChannelPipeline pipeline = channel.pipeline();
    if (invokeLater) {
        Runnable task = fireChannelWritabilityChangedTask;
        if (task == null) {
            fireChannelWritabilityChangedTask = task = new Runnable() {
                @Override
                public void run() {
                    pipeline.fireChannelWritabilityChanged();
                }
            };
        }
        channel.eventLoop().execute(task);
    }
}
```

---

```

    } else {
        pipeline.fireChannelWritabilityChanged();
    }
}

```

当消息发送完成后，对低水位进行判断，如果当前积压的待发送字节数到达或者低于低水位，则修改 Channel 状态为可写，并发送通知事件，代码如下：

```

private void decrementPendingOutboundBytes(long size, boolean invokeLater,
boolean notifyWritability) {
    if (size == 0) {
        return;
    }
    long newWriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this,
-size);
    if (notifyWritability && newWriteBufferSize < channel.config().
getWriteBufferLowWaterMark()) {
        setWritable(invokeLater);
    }
}
}

```

利用 Netty 的高低水位机制，可以防止在发送队列处于高水位时继续发送消息，导致积压更严重，甚至发生内存泄漏。在业务中合理利用 Netty 的高低水位机制，可以提升系统的可靠性。

## 5.3 总结

本章通过发送队列积压案例，对 Netty 的消息发送原理和源码进行了深入讲解，熟悉了 Netty 的发送队列工作机制、高低水位机制等，就可以在实际项目中更好地利用这些功能，提升基于 Netty 构建的通信框架的可靠性。

## 第 6 章

---

# API 网关高并发压测性能波动案例

当前很多 Java 版的 API 网关选择基于 Netty 构建，例如 Netflix 开源的 Zuul2，API 网关负责统一的流量接入、安全控制、鉴权、流控和消息路由转发等，对性能要求非常苛刻，在高并发场景下，如果处理不当，很有可能导致内存泄漏、性能波动等问题。

本章通过一个 API 网关的性能波动案例，让大家学习如何基于 Netty 构建高性能的 API 网关，以及如何处理好性能相关的编程细节。

### 6.1 高并发压测性能波动问题

---

平台基于 Netty 构建了高性能的 API 网关，支持 HTTP、Restful、TCP 等的接入和消息路由。当性能压测到 4000 QPS 一段时间后，发现性能急剧下降，最低时到 0，停止压测一段时间，系统恢复，再压测一段时间，用户并发量大了之后，性能又急剧下降，形成周期性波动，吞吐量非常不稳定。

#### 6.1.1 故障场景模拟

---

为了方便分析，对 API 网关的代码做简化处理，模拟代码如下：



```

public class ApiGatewayServerHandler extends ChannelInboundHandlerAdapter {
    ExecutorService executorService = Executors.newFixedThreadPool(8);

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ctx.write(msg);
        char [] req = new char[64 * 1024];
        executorService.execute(() ->
        {
            char [] dispatchReq = req;
            //简单处理之后，转发请求消息到后端服务，后续代码省略
            try
            {
                //模拟业务逻辑处理，耗时 0.5ms
                TimeUnit.MICROSECONDS.sleep(500);
            } catch (Exception e)
            {
                e.printStackTrace();
            }
        });
    }
}

```

压测一段时间之后，发现内存和 CPU 占用飙升，同时 QPS 下降，监控数据如图 6-1 所示。

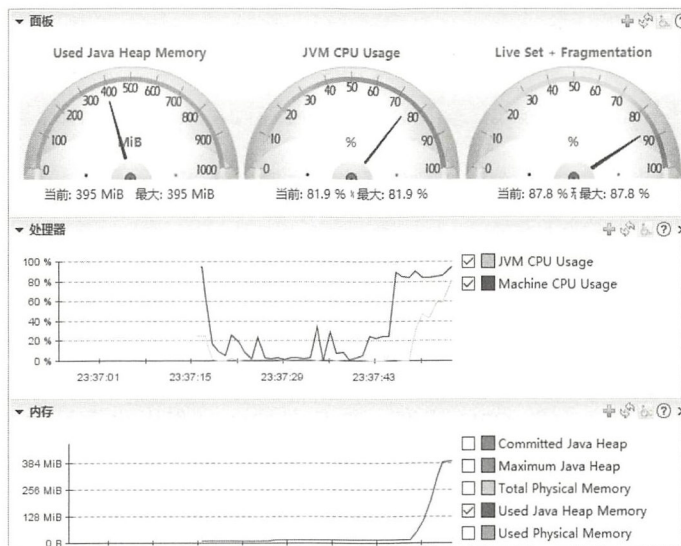


图 6-1 监控数据

客户端停止压测一段时间，CPU 占用降低，内存占用恢复正常，如图 6-2 所示。

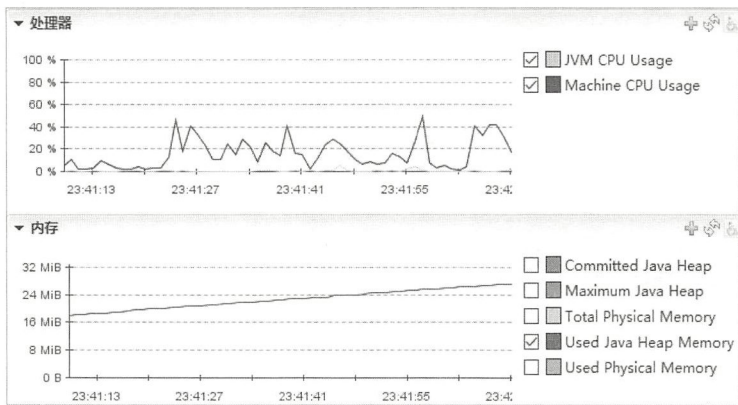


图 6-2 停止压测一段时间系统恢复正常

在压测过程中，当吞吐量下降、CPU 占用飙升时，采集线程堆栈，发现 GC 线程占用大量 CPU 资源，堆栈如图 6-3 所示（需要采集线程 CPU 占用相关数据，此处省略）。

```
"VM Thread" os_prio=2 tid=0x0000000013da8800 nid=0x33d8 runnable
"GC task thread=0 (ParallelGC)" os_prio=0 tid=0x0000000002988000 nid=0x3524 runnable
"GC task thread=1 (ParallelGC)" os_prio=0 tid=0x0000000002989800 nid=0x39c runnable
"GC task thread=2 (ParallelGC)" os_prio=0 tid=0x000000000298b000 nid=0x2b94 runnable
"GC task thread=3 (ParallelGC)" os_prio=0 tid=0x000000000298c800 nid=0x3ad8 runnable
"VM Periodic Task Thread" os_prio=2 tid=0x0000000015204000 nid=0x1804 waiting on condition
```

图 6-3 GC 线程占用大量 CPU 资源

## 6.1.2 性能波动原因定位

通过监控数据分析，发现性能波动与内存占用情况强相关，当内存占用比较高时，GC 线程忙于内存回收，抢占大量 CPU 资源，而且在 GC 过程中也会导致应用线程暂停（不同的 GC 收集器，暂停策略不同），最终造成吞吐量急剧下降。但是当停止压测或者并发请求量降低之后，服务端还可以恢复，说明并不存在真正的内存忘记释放导致的泄漏问题。

Dump 内存堆栈，通过 Mat 工具进行 Top 内存占用情况分析，内存分布总体概况如图 6-4 所示。

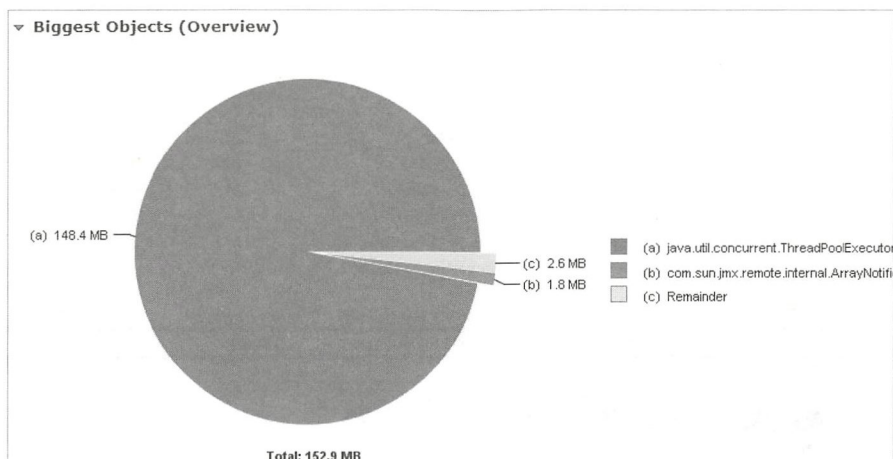


图 6-4 内存分布总体概况

JDK 的线程池是内存占用大户，切换到类实例图，看看究竟哪些对象占用内存多，如图 6-5 所示。

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	13,417	157,491,432	>= 157,491,432
io.netty.cases.chapter.demo6.ApiG...	1,195	19,120	>= 156,669,280
java.util.concurrent.ThreadPoolExe...	4	288	>= 155,651,840
java.util.concurrent.LinkedBlocking...	4	192	>= 155,649,728
java.util.concurrent.LinkedBlocking...	1,191	28,584	>= 155,649,032
java.lang.Object[]	3,684	609,384	>= 2,790,424
com.sun.jmx.remote.internal.Array...	1	64	>= 1,868,176
com.sun.jmx.remote.internal.Array...	1	32	>= 1,867,720
com.sun.jmx.remote.internal.Array...	198	4,752	>= 1,863,664

图 6-5 内存占用实例图

char 数组共 13417 个实例，占用 157MB 内存，通过 Dominator Tree 查看引用关系，发现 char 数组被 JDK 线程池的 LinkedBlockingQueue 引用，LinkedBlockingQueue 是一个链表队列，引用关系如图 6-6 所示。

结合代码分析发现，API 网关每次收到请求消息，无论请求消息大还是小，哪怕只有 100 字节，都会构造一个默认为 64KB 的 char 数组，用于处理和转发请求消息。如果后端转发消息较慢，就会导致任务队列积压，由于每个任务（Runnable，此处为 Lambda 表达式）持有一个 64KB 的 char 数组，所以积压多了就会转移到老年代，一旦触发老年代 GC，

耗时较长，就会导致系统吞吐量降为 0。

java.util.concurrent.ThreadPoolExecutor @ 0xe04c5bc0	72	155,649,872	96.29%
java.util.concurrent.LinkedBlockingQueue @ 0xe04c5c48	48	155,649,200	96.29%
java.util.concurrent.LinkedBlockingQueue\$Node @ 0xed2fe938	24	155,517,832	96.21%
java.util.concurrent.LinkedBlockingQueue\$Node @ 0xed2fe960	24	155,517,808	96.21%
java.util.concurrent.LinkedBlockingQueue\$Node @ 0xed2fe988	24	155,386,680	96.12%
java.util.concurrent.LinkedBlockingQueue\$Node @ 0xed2fe9b0	24	155,255,552	96.04%
java.util.concurrent.LinkedBlockingQueue\$Node @ 0xed2fe9d8	24	155,124,424	95.96%
io.netty.cases.chapter.demo6.ApiGatewayServerHandler\$\$Lambda\$1	16	131,104	0.08%
Total: 2 entries			
io.netty.cases.chapter.demo6.ApiGatewayServerHandler\$\$Lambda\$2	16	131,104	0.08%
char[65536] @ 0xef166170 \u0000\u0000\u0000\u0000\u0000\u0000	131,088	131,088	0.08%
Total: 2 entries			
io.netty.cases.chapter.demo6.ApiGatewayServerHandler\$\$Lambda\$3	16	131,104	0.08%
char[65536] @ 0xef146160 \u0000\u0000\u0000\u0000\u0000\u0000	131,088	131,088	0.08%
Total: 2 entries			

图 6-6 对象引用关系

由于当前网关平台转发的请求报文都较小（1KB 左右），因此对原有代码进行优化，按照请求消息大小来初始化 char 数组，代码修改如下（ApiGatewayServerHandler 类）：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    //代码省略
    char [] req = new char[((ByteBuf)msg).readableBytes()];
    //代码省略
}
```

优化之后，API 网关运行平稳，CPU 和内存使用情况恢复正常，如图 6-7 所示。

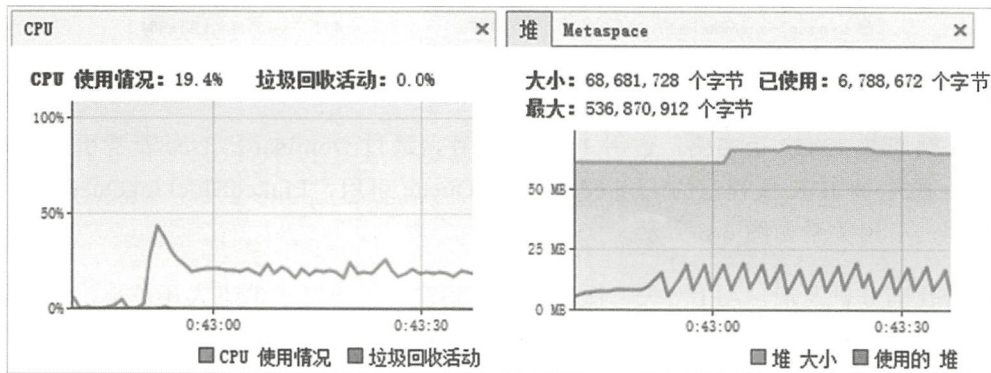


图 6-7 优化之后的 CPU 和内存资源占用监控



需要指出的是，这里给出的代码是模拟示例代码，不同 CPU、内存配置的机器运行结果可能存在差异，重要的是理解其中的原理：当内存申请和释放速度能够相对平衡时，系统运行就平稳，反之如果老年代经常发生堆积，频繁触发 CMS GC 甚至 Full GC，则系统就很难稳定运行。

### 6.1.3 主动内存泄漏定位法

示例代码比较简单，char 数组的引用关系通过 Dump 堆栈分析甚至阅读源码就能轻易获得。在实际的 API Gateway 中，往往同时运行多个业务 API、同时还会打印日志、采集调用链信息等，对象的引用关系异常复杂，很难通过简单的代码分析就得出引用关系。

另外，由于生产环境系统内存配置得比较大（例如 32GB），对于一次新生代 GC 无法回收的对象，特别是连续的大块内存（最典型的的就是 byte 数组和 char 数组等），超过存活的 age 阈值会被晋升为老年代，引起老年代的缓慢积压。如果积压速度比较慢，需要测试数小时甚至几天才能重现一次问题，吞吐量降为 0 时，Dump 内存信息，有时候会找不到 GC 引用关系（例如 Dump 内存堆栈时触发 Full GC 清除了引用关系），如图 6-8 所示。

[illegible]

图 6-8 找不到对象引用关系

好不容易重现一次问题，采集了故障时的内存堆栈，却没有分析出引用关系，确实令人沮丧。

对于类似问题，可以采用主动 OOM 法，具体策略为：将最大堆内存尽量调小，让系统在更短的时间内发生 OOM 异常，多采集几次 OOM 异常时的内存堆栈信息进行对比，同时通过 OOM 日志，结合 Dump 堆栈，就更容易找到对象引用关系。

在实际业务中进行性能压测时，案例中的 API Gateway 每 6 小时才会发生一次问题，



采用主动 OOM 定位法后，每 10 分钟就发生一次 OOM，通过多次采集内存堆栈信息进行对比分析，很快就确定了问题的根源。

#### 6.1.4 网关类产品的优化建议

---

网关类产品的主要功能就是消息的预处理和转发，请求和响应对象都是“朝生夕灭”类型的，在高并发场景下，一定要防止不合理的内存申请，具体措施如下。

（1）内存按需分配。不要一次性申请较大的内存来保存较小的消息，造成内存空间浪费，引发频繁 GC 问题。

（2）不要频繁地创建和释放对象。会增加 GC 的负担，降低系统的吞吐量，可以采用内存池等机制优化内存的申请和释放。

（3）减少对象拷贝。对于透传类的消息，尽量把涉及业务逻辑处理的字段放入 Header，不要对 Body 做解码，直接透传到后端服务即可。这样可以大幅减少内存的申请和对象拷贝，降低内存占用，提升性能。

（4）流控机制必不可少。除了客户端并发连接数流控、QPS 流控，还需要针对内存占用等指标做流控，防止业务高峰期的 OOM。

## 6.2 Netty 消息接入内存申请机制

---

作为高性能的 NIO 通信框架，Netty 对于协议消息接入的内存申请及动态扩容的处理策略，值得我们借鉴和学习。下面分析 Netty 消息接入的内存申请方式，以及当 ByteBuffer 容量不足时的动态扩容机制。对于 API 网关类产品，尤其有借鉴意义。

### 6.2.1 消息接入的内存分配原理和源码分析

---

当 SocketChannel 有消息读取时，需要预先分配一个 ByteBuffer 接收消息，但是由于消息尚未被读取，无法知道需要申请一个多大的 ByteBuffer。如果小了就需要重新申请一

个大的 `ByteBuffer`，将原来已经读取到较小 `ByteBuffer` 的 `byte` 数组拷贝到新申请的 `ByteBuffer`，同时释放老的 `ByteBuffer`，这会增加一次内存拷贝操作，而且申请了两个 `ByteBuffer`，更占用内存。如果过大，则会导致内存资源的浪费。下面分析一下 Netty 的解决策略。

Netty NIO 消息读取入口是 `NioByteUnsafe` 的 `read` 方法，代码如下：

---

```
public final void read() {
    final ChannelConfig config = config();
    //代码省略
    final ByteBufAllocator allocator = config.getAllocator();
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);
    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            byteBuf = allocHandle.allocate(allocator);

```

---

默认通过 `AdaptiveRecvByteBufAllocator` 来进行内存分配，它的成员变量定义如下：

---

```
public class AdaptiveRecvByteBufAllocator extends
DefaultMaxMessagesRecvByteBufAllocator {
    static final int DEFAULT_MINIMUM = 64;
    static final int DEFAULT_INITIAL = 1024;
    static final int DEFAULT_MAXIMUM = 65536;
    private static final int INDEX_INCREMENT = 4;
    private static final int INDEX_DECREMENT = 1;
    private static final int[] SIZE_TABLE;
}

```

---

它分别定义了 3 个系统默认值：最小缓冲区长度为 64 字节、初始容量为 1024 字节、最大容量为 65536 字节。还定义了两个动态调整容量的步进参数：扩张的步进索引为 4、收缩的步进索引为 1。

最后，定义了长度的向量表 `SIZE_TABLE` 并初始化，初始值如图 6-9 所示。

0-->16	1-->32	2-->48	3-->64	4-->80	5-->96	6-->112	7-->128	8-->144	9-->160
10-->176	11-->192	12-->208	13-->224	14-->240	15-->256	16-->272	17-->288	18-->304	
19-->320	20-->336	21-->352	22-->368	23-->384	24-->400	25-->416	26-->432	27-->448	
28-->464	29-->480	30-->496	31-->512	32-->1024	33-->2048	34-->4096	35-->8192	36-->16384	
37-->32768	38-->65536	39-->131072	40-->262144	41-->524288	42-->1048576	43-->2097152	44-->4194304	45-->8388608	
46-->16777216	47-->33554432	48-->67108864	49-->134217728	50-->268435456	51-->536870912	52-->1073741824			

图 6-9 `SIZE_TABLE` 的初始值

向量数组的每个值都对应一个 Buffer 容量，当容量小于 512 字节的时候，由于缓冲区已经比较小，需要降低步进值，容量每次下调的幅度要小些；当容量大于 512 字节时，说明需要解码的消息码流比较大，这时采用调大步进幅度的方式减小动态扩张的频率，所以它采用 512 字节的倍数进行扩张。接下来重点分析 `AdaptiveRecvByteBufAllocator` 的相关方法。

`getSizeTableIndex(final int size)`代码如下：

---

```
private static int getSizeTableIndex(final int size) {
    for (int low = 0, high = SIZE_TABLE.length - 1;;) {
        if (high < low) {
            return low;
        }
        if (high == low) {
            return high;
        }
        int mid = low + high >>> 1;
        int a = SIZE_TABLE[mid];
        int b = SIZE_TABLE[mid + 1];
        if (size > b) {
            low = mid + 1;
        } else if (size < a) {
```

---

```

        high = mid - 1;
    } else if (size == a) {
        return mid;
    } else {
        return mid + 1;
    }
}
}

```

---

根据容量 Size 查找容量向量表对应的索引——这是典型的二分查找法，由于它的算法比较简单，此处不再展开说明。

下面继续分析下它的内部静态类 HandleImpl，首先，还是看它的成员变量定义：

```

private final class HandleImpl extends MaxMessageHandle {
    private final int minIndex;
    private final int maxIndex;
    private int index;
    private int nextReceiveBufferSize;
    private boolean decreaseNow;
}

```

---

它有 5 个成员变量，分别是对应向量表的最小索引、最大索引、当前索引、下一次预分配的 Buffer 大小，以及是否立即执行容量收缩操作。

接下来重点分析它的 record(int actualReadBytes)方法：当 NioSocketChannel 执行完读操作，会计算本次轮询读取的总字节数，它就是参数 actualReadBytes，执行 record 方法，根据实际读取的字节数对 ByteBuf 进行动态伸缩，代码如下：

```

private void record(int actualReadBytes) {
    if (actualReadBytes <= SIZE_TABLE[max(0, index - INDEX_DECREMENT - 1)]) {
        if (decreaseNow) {
            index = max(index - INDEX_DECREMENT, minIndex);
            nextReceiveBufferSize = SIZE_TABLE[index];
            decreaseNow = false;
        }
    }
}

```

```

        } else {
            decreaseNow = true;
        }
    } else if (actualReadBytes >= nextReceiveBufferSize) {
        index = min(index + INDEX_INCREMENT, maxIndex);
        nextReceiveBufferSize = SIZE_TABLE[index];
        decreaseNow = false;
    }
}

```

---

首先，对当前索引做步进缩减，然后获取收缩后索引对应的容量，与实际读取的字节数进行比对，如果发现实际读取的字节数小于收缩后的容量，则重新对当前索引进行赋值，取收缩后的索引和最小索引中的较大者作为新的索引。然后，为下一次缓冲区容量分配赋值——新的索引对应容量向量表中的容量。相反，如果实际读取的字节数大于之前预分配的初始容量，则说明实际分配的容量不足，需要动态扩张。重新计算索引，选取“当前索引+扩张步进”和最大索引中的较小作为当前索引值，然后对下次缓冲区的容量值进行重新分配，完成缓冲区容量的动态扩张。

通过上述分析得知，AdaptiveRecvByteBufAllocator 根据本次读取的实际字节数对下次接收缓冲区的容量进行动态调整。

Netty 的动态缓冲区分配器优点如下。

(1) Netty 作为一个通用的 NIO 框架，不能对用户的应用场景进行假设，可以使用它做流式计算，也可以用它做 RCP 框架，不同的应用场景，传输的码流大小千差万别，无论初始化时分配的是 32KB 还是 1MB，都会随着应用场景的变化而变得不合适。因此，Netty 根据上次实际读取的码流大小对下次的接收 Buffer 缓冲区进行预测和调整，能够最大限度地满足不同行业的应用场景的需要。

(2) 综合性能更高。分配容量过大会导致内存占用开销增加，后续的 Buffer 处理性能下降；容量过小需要频繁地内存扩张来接收大的请求消息，同样会导致性能下降。

(3) 更节约内存。假如通常情况请求消息大小平均值为 1MB 左右，接收缓冲区大小为 1.2MB，突然某个客户发送了一个 10MB 的附件，接收缓冲区扩张为 10MB 以读取该附件，如果缓冲区不能收缩，每次缓冲区创建都会分配 10MB 的内存，但是后续所有的消息都是 1MB



左右的，这样会导致内存的浪费，如果并发客户端过多，可能会导致内存溢出宕机。

## 6.2.2 Netty ByteBuf 的动态扩容原理和源码分析

以 `UnpooledHeapByteBuf` 为例进行说明，它的 `capacity(int newCapacity)` 方法如下（`UnpooledHeapByteBuf` 类）：

---

```

public ByteBuf capacity(int newCapacity) {
    checkNewCapacity(newCapacity);
    int oldCapacity = array.length;
    byte[] oldArray = array;
    if (newCapacity > oldCapacity) {
        byte[] newArray = allocateArray(newCapacity);
        System.arraycopy(oldArray, 0, newArray, 0, oldArray.length);
        setArray(newArray);
        freeArray(oldArray);
    } else if (newCapacity < oldCapacity) {
        byte[] newArray = allocateArray(newCapacity);
        int readerIndex = readerIndex();
        if (readerIndex < newCapacity) {
            int writerIndex = writerIndex();
            if (writerIndex > newCapacity) {
                writerIndex(writerIndex = newCapacity);
            }
            System.arraycopy(oldArray, readerIndex, newArray, readerIndex,
writerIndex - readerIndex);
        } else {
            setIndex(newCapacity, newCapacity);
        }
        setArray(newArray);
        freeArray(oldArray);
    }
    return this;
}

```

---

方法入口首先对新容量进行合法性校验，如果大于容量上限或者小于 0，则抛出 `IllegalArgumentException` 异常。判断新的容量值是否大于当前的缓冲区容量，如果大于则需要进行动态扩展，通过 `byte[] newArray = new byte[newCapacity]` 创建新的缓冲区字节数组，然后通过 `System.arraycopy` 进行内存复制，将旧的字节数组复制到新创建的字节数组，最后调用 `setArray` 替换旧的字节数组，代码如下：

---

```
private void setArray(byte[] initialArray) {  
    array = initialArray;  
    tmpNioBuf = null;  
}
```

---

需要指出的是，当动态扩容完成后，需要将原来的视图 `tmpNioBuf` 设置为空。

如果新的容量小于当前的缓冲区容量，不需要动态扩展，但是需要截取当前缓冲区创建一个新的子缓冲区，具体的算法为：首先判断读索引是否小于新的容量值，如果是，进一步判断写索引是否大于新的容量值，如果是，则将写索引设置为新的容量值（防止越界）。更新完写索引，通过内存复制 `System.arraycopy` 将当前可读的字节数组复制到新创建的子缓冲区中。如果新的容量值小于读索引，说明没有可读的字节数组需要复制到新创建的缓冲区，将读写索引设置为新的容量值即可。最后调用 `setArray` 方法替换原来的字节数组。扩容后调用 `freeArray` 释放原来的 `byte` 数组 `oldArray`。

## 6.3 总结

对于高并发接入的 API 网关类产品，需要谨慎处理消息的内存申请和释放，减少不必要的申请（例如透传类场景），同时要防止内存空间的浪费。借鉴 Netty 请求消息读取的内存申请策略和动态扩容机制，并用在实际项目中，可以得到较大的性能提升。

## 第 7 章

---

# Netty ChannelHandler 并发安全案例

ChannelHandler 是 Netty 中使用最广的接口，Netty 提供了大量内置的 ChannelHandler 实现类，包括编解码、SSL、日志打印和流量整形等。用户通过实现 ChannelHandler 接口，来接收和发送业务消息，并进行业务逻辑处理。

ChannelHandler 的一端是 Netty NIO 线程，另一端则是业务线程池，在多线程并发场景下理解 ChannelHandler 的并发安全性很重要，如果使用不当，会产生性能和并发安全问题。

## 7.1 Netty ChannelHandler 并发安全问题

---

业务有一个非线程安全的类 ThreadUnsafeClass，这个类会在业务 ChannelHandler 的 channelRead 方法中被调用。下面这样的调用方法在多线程环境下安全吗？

```
public class ServiceHandler extends ChannelInboundHandlerAdapter {  
    private ThreadUnsafeClass unsafe = new ThreadUnsafeClass(); // 非线程安全  
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
```

```

//疑问：下面的代码是否正确？
unsafe.doSomething(ctx, msg);
}
//后续代码省略

```

---

### 7.1.1 串行执行的 ChannelHandler

如果 ChannelHandler 是非共享的，则它就是线程安全的，因为：当链路完成初始化时会创建 ChannelPipeline，每个 Channel 对应一个 ChannelPipeline 实例，业务的 ChannelHandler 会被实例化并加入 ChannelPipeline 执行。由于某个 Channel 只能被特定的 NioEventLoop 线程执行，因此 ChannelHandler 不会被并发调用，不用考虑线程安全问题，相关代码示例如下：

---

```

.handler(new ChannelInitializer<SocketChannel>()) {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(
            //每个链路都有自己对应的业务 Handler 实例，不共享
            new NoThreadSecurityClientHandler());
    }
});
}
//后续代码省略

```

---

业务可能会多线程调用 ChannelHandlerContext 或者 Channel 的 write 方法，这会不会导致多个业务线程并发调用 ChannelHandler 呢？通过源码分析发现，不会产生并发安全问题。在执行 write 操作时，判断是否是下一个要执行 write 操作的 AbstractChannelHandlerContext 的 EventExecutor 线程，如果不是则将 write 操作封装成 AbstractWriteTask 放入线程任务队列异步执行，原调用线程返回。如果业务的 ChannelHandler 没有指定（通常不需要指定）EventExecutor 线程，则使用的就是消息读写对应的 NioEventLoop 线程。由此看见，即便多个业务线程并发调用某个 Channel，也不会产生多个线程并发访问业务 ChannelHandler 的问题。源码如下（AbstractChannelHandlerContext 类）：



```
private void write(Object msg, boolean flush, ChannelPromise promise) {  
    AbstractChannelHandlerContext next = findContextOutbound();  
    final Object m = pipeline.touch(msg, next);  
    EventExecutor executor = next.executor();  
    if (executor.inEventLoop()) {  
        if (flush) {  
            next.invokeWriteAndFlush(m, promise);  
        } else {  
            next.invokeWrite(m, promise);  
        }  
    } else {  
        AbstractWriteTask task;  
        if (flush) {  
            task = WriteAndFlushTask.newInstance(next, m, promise);  
        } else {  
            task = WriteTask.newInstance(next, m, promise);  
        }  
        safeExecute(executor, task, promise, m);  
    }  
}
```

//后续代码省略

串行执行的 ChannelHandler 的工作原理如图 7-1 所示。

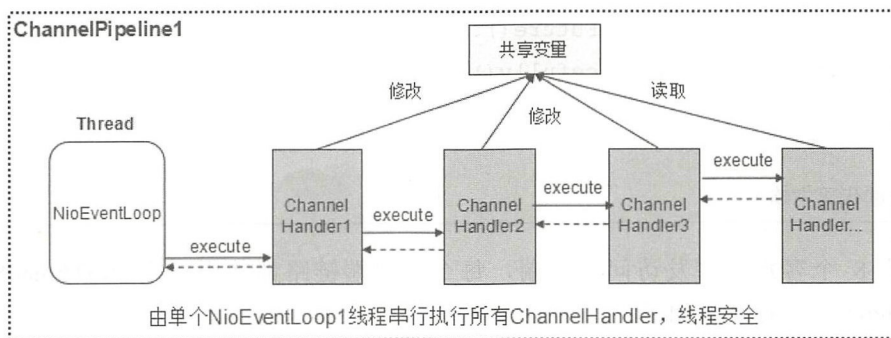


图 7-1 串行执行的 ChannelHandler 的工作原理





## Netty 进阶之路：跟着案例学 Netty

有的读者可能会有疑问，如果只初始化一次业务 `ChannelHandler`，然后加到多个 `Channel` 的 `ChannelPipeline`，由于不同的 `Channel` 可能绑定不同的 `NioEventLoop` 线程，这样 `ChannelHandler` 就可能会被多个 I/O 线程访问，存在并发安全风险。那实际结果又如何呢？我们可以测试一下，代码修改如下（`NoThreadSecurityClient` 类）：

---

```
public void connect() throws Exception
{
    EventLoopGroup group = new NioEventLoopGroup(8);
    Bootstrap b = new Bootstrap();
    ChannelHandler clientHanlder = new NoThreadSecurityClientHandler();
    b.group(group)
      .channel(NioSocketChannel.class)
      .option(ChannelOption.TCP_NODELAY, true)
      .handler(new ChannelInitializer<SocketChannel>() {
          @Override
          public void initChannel(SocketChannel ch) throws Exception {
              ch.pipeline().addLast(clientHanlder);
          }
      });
    ChannelFuture f = null;
    for(int i = 0; i < 8; i++)
    {
        f = b.connect(HOST, PORT).sync();
    }
    f.channel().closeFuture().sync();
    group.shutdownGracefully();
}

//后续代码省略
```

---

模拟 8 个客户端并发访问服务端，每个客户端链路都共用一个 `NoThreadSecurityClientHandler`，运行结果如下：

---

```
io.netty.channel.ChannelPipelineException:
```

---

`io.netty.cases.chapter.demo7.NoThreadSecurityClientHandler` is not a `@Sharable` handler, so can't be added or removed multiple times.

```

    at io.netty.channel.DefaultChannelPipeline.checkMultiplicity
(DefaultChannelPipeline.java:625)
    at io.netty.channel.DefaultChannelPipeline.addLast
(DefaultChannelPipeline.java:208)
    at io.netty.channel.DefaultChannelPipeline.addLast
(DefaultChannelPipeline.java:409)
    at io.netty.channel.DefaultChannelPipeline.addLast
(DefaultChannelPipeline.java:396)
    at io.netty.cases.chapter.demo7.NoThreadSecurityClient$1.initChannel
(NoThreadSecurityClient.java:48)
    at io.netty.cases.chapter.demo7.NoThreadSecurityClient$1.initChannel
(NoThreadSecurityClient.java:45)
    at io.netty.channel.ChannelInitializer.initChannel
(ChannelInitializer.java:115)
    at io.netty.channel.ChannelInitializer.handlerAdded
(ChannelInitializer.java:107)

```

---

我们发现，当把同一个 `ChannelHandler` 加入多个 `ChannelPipeline` 时会发生异常，相关代码如下：

```

private static void checkMultiplicity(ChannelHandler handler) {
    if (handler instanceof ChannelHandlerAdapter) {
        ChannelHandlerAdapter h = (ChannelHandlerAdapter) handler;
        if (!h.isSharable() && h.added) {
            throw new ChannelPipelineException(
                h.getClass().getName() +
                " is not a @Sharable handler, so can't be added or removed
multiple times.");
        }
        h.added = true;
    }
}

```

---



如果 `ChannelHandler` 不是共享的，重复向 `ChannelPipeline` 添加时就会抛出 `ChannelPipelineException` 异常，添加失败。所以非共享的同一个 `ChannelHandler` 实例不能被重复加入多个 `ChannelPipeline` 或者被多次加入某一个 `ChannelPipeline`。

## 7.1.2 跨链路共享的 `ChannelHandler`

如果某个 `ChannelHandler` 需要全局共享，则通过 `Sharable` 注解就可以被添加到多个 `ChannelPipeline`，示例代码如下：

---

```
@ChannelHandler.Sharable
```

```
public class SharableClientHandler extends ChannelInboundHandlerAdapter {  
    //后续代码省略  
}
```

---

对上一节中的代码做修改，代码如下：

---

```
@ChannelHandler.Sharable
```

```
public void connect() throws Exception  
{  
    EventLoopGroup group = new NioEventLoopGroup(8);  
    Bootstrap b = new Bootstrap();  
    ChannelHandler clientHanlder = new SharableClientHandler();  
    //后续代码省略  
}
```

---

运行结果如图 7-2 所示，共享的 `ChannelHandler` 可以被加入多个 `ChannelPipeline`。

```
Client receive msg : PooledUnsafeDirectByteBuf(ridx: 0, widx: 256, cap: 272)  
Client receive msg : PooledUnsafeDirectByteBuf(ridx: 0, widx: 256, cap: 272)  
Client receive msg : PooledUnsafeDirectByteBuf(ridx: 0, widx: 256, cap: 272)
```

图 7-2 `Sharable` 注解的 `ChannelHandler` 运行结果

当 `ChannelHandler` 被添加到多个 `ChannelPipeline`，就会面临多线程并发访问问题，需要 `ChannelHandler` 保证自身的线程安全，例如通过原子类、读写锁等方式对数据做并发保护。如果加锁，可能会阻塞 `NioEventLoop` 线程，所以 `Sharable` 注解的 `ChannelHandler` 要慎用。





例如, ChannelHandler 是共享的, 但它自身也不是线程安全的, 这就会导致并发问题, 客户端代码示例如下 (SharableClientHandler 类):

```
@ChannelHandler.Sharable
public class SharableClientHandler extends ChannelInboundHandlerAdapter {
    int counter = 0;
    //代码省略
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf req = (ByteBuf)msg;
        if (counter ++ < 10000)
            ctx.write(msg);
    }
}
```

counter 是成员变量, 代码 if(counter ++ < 10000)是非线程安全的, 因此它无法保证精确地控制发送次数, 如图 7-3 所示。

```
Server receive client message : 10139
Server receive client message : 10140
Server receive client message : 10142
Server receive client message : 10141
Server receive client message : 10143
Server receive client message : 10144
Server receive client message : 10145
Server receive client message : 10146
```

图 7-3 非线程安全的 ChannelHandler 测试结果

对 SharableClientHandler 类进行修改, 将其实现改为线程安全的, 代码如下:

```
@ChannelHandler.Sharable
public class SharableClientHandler extends ChannelInboundHandlerAdapter {
    AtomicInteger counter = new AtomicInteger(0);
    //代码省略
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf req = (ByteBuf)msg;
```



```
        if (counter.getAndIncrement() < 10000)
            ctx.write(msg);
    }
}
```

由于将 `SharableClientHandler` 修改成了线程安全类，所以在并发访问时依然能够保证逻辑的正确性，如图 7-4 所示。在初始化链路时 8 个客户端共发了 8 条消息，因此服务端接收的总消息数为 10008，与服务端统计的结果一致，说明客户端的并发控制准确无误。

```
Server receive client message : 10001
Server receive client message : 10002
Server receive client message : 10003
Server receive client message : 10004
Server receive client message : 10005
Server receive client message : 10006
Server receive client message : 10007
Server receive client message : 10008
```

图 7-4 线程安全的 `ChannelHandler` 测试结果

当用户使用 `ChannelHandler.Sharable` 注解时，一定要谨慎，尽量少用，如果确实需要使用，则需要自己保证 `ChannelHandler` 的线程安全性，否则将产生并发问题，程序无法正常运行。

### 7.1.3 `ChannelHandler` 的并发陷阱

用户自定义的 `ChannelHandler` 有两种场景需要考虑并发安全。

(1) 通过 `Sharable` 注解，多个 `ChannelPipeline` 共享的 `ChannelHandler`，它将被多个 `NioEventLoop` 线程（通常用户创建的 `NioEventLoopGroup` 线程数>1）并发访问，如图 7-5 所示。



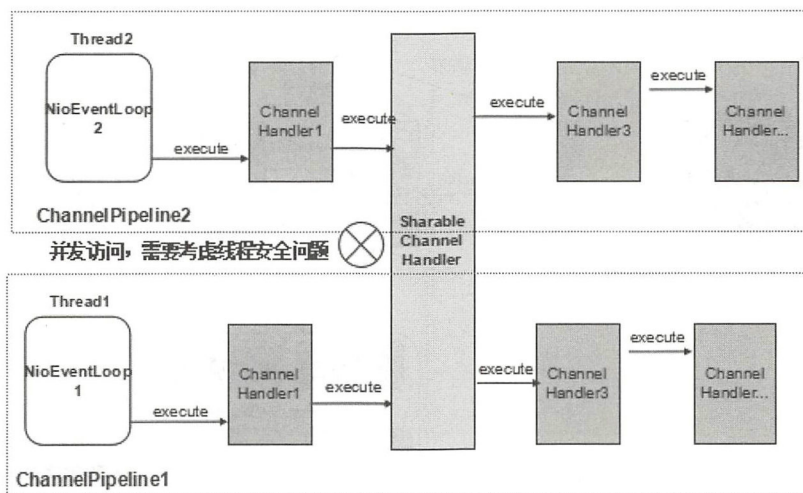


图 7-5 多个 ChannelPipeline 共享的 ChannelHandler 原理

在这种场景下，用户需要保证 ChannelHandler 共享的合理性，同时需要自己保证它的并发安全性，尽量通过原子类等方式降低锁的开销，防止阻塞 NioEventLoop 线程。

(2) ChannelHandler 没有共享，但是在用户的 ChannelPipeline 中的一些 ChannelHandler 绑定了新的线程池，这样 ChannelPipeline 的 ChannelHandler 就会被异步执行，如图 7-6 所示。



图 7-6 ChannelHandler 绑定线程池相关接口

在多线程异步执行过程中，如果某 ChannelHandler 的成员变量共享给其他 ChannelHandler，那么被多个线程并发访问和修改就存在并发安全问题，如图 7-7 所示。

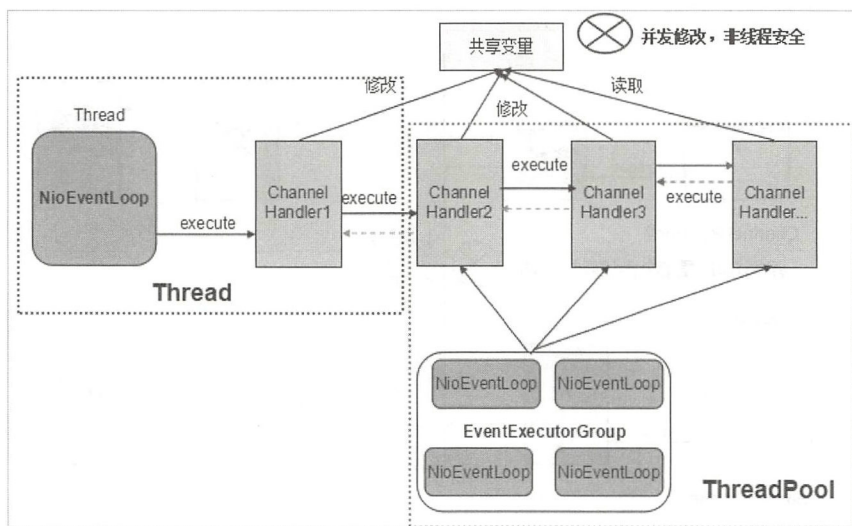


图 7-7 ChannelHandler 并发安全问题原理

## 7.2 Netty ChannelHandler 工作机制

Netty 的 ChannelHandler 工作原理与 Servlet Filter 机制一致,它将 Channel 的数据管道抽象为 ChannelPipeline,消息在 ChannelPipeline 中流动和传递。ChannelPipeline 持有 I/O 事件拦截器 ChannelHandler 的链表,由 ChannelHandler 对 I/O 事件进行拦截和处理,可以方便地通过新增和删除 ChannelHandler 来实现不同的业务逻辑定制,不需要对已有的 ChannelHandler 进行修改,能够实现对修改封闭和对扩展的支持。对于用户来说,基于 Netty 的编程就是开发和组装各种系统与业务 ChannelHandler,实现业务逻辑处理。

### 7.2.1 职责链 ChannelPipeline 原理和源码分析

ChannelPipeline 是 ChannelHandler 的编排管理容器,它内部维护了一个 ChannelHandler 的链表和迭代器,可以方便地实现 ChannelHandler 的查找、添加、替换和删除。

消息经过 ChannelPipeline 在 ChannelHandler 中的处理流程如图 7-8 所示。



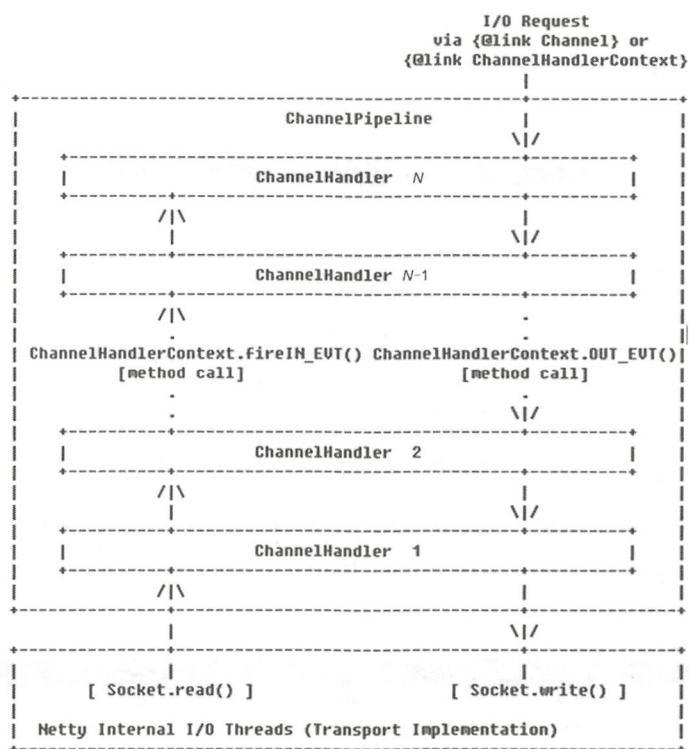


图 7-8 消息经过 ChannelPipeline 在 ChannelHandler 中的处理流程

它的工作原理如下。

(1) 底层的 SocketChannel read()方法读取 ByteBuf, 触发 ChannelRead 事件, 由 I/O 线程 NioEventLoop 调用 ChannelPipeline 的 fireChannelRead(Object msg)方法, 将消息 (ByteBuf) 传到 ChannelPipeline。

(2) 消息依次被 HeadContext、ChannelHandler1、ChannelHandler2……TailHandler 拦截和处理, 在这个过程中, 任何 ChannelHandler 都可以中断当前的流程, 结束消息的传递。

(3) 调用 ChannelHandlerContext 的 write()方法发送消息, 消息从 TailContext 开始, 途经 ChannelHandlerN……ChannelHandler1、HeadHandler, 最终被添加到消息发送缓冲区等待刷新和发送, 在此过程中也可以中断消息的传递, 例如当编码失败时, 就需要中断流程, 构造异常的 Future 返回。

Netty 中的事件分为 inbound 事件和 outbound 事件。inbound 事件通常由 I/O 线程触发,





例如 TCP 链路建立事件、链路关闭事件、读事件、异常通知事件，用户通过实现 `ChannelInboundHandler` 接口完成消息的读取、反序列化、鉴权等工作，其接口定义如图 7-9 所示。

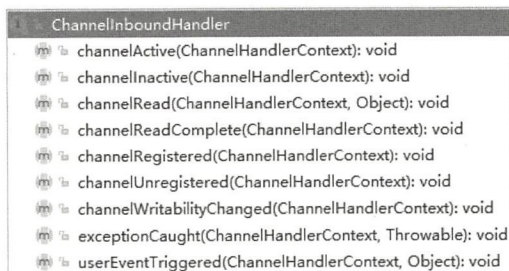


图 7-9 `ChannelInboundHandler` 接口定义

`outbound` 事件通常是由用户主动发起的网络 I/O 操作，例如用户发起的连接操作、绑定操作、消息发送操作等，通过实现 `ChannelOutboundHandler` 来拦截并处理相关事件，其接口定义如图 7-10 所示。

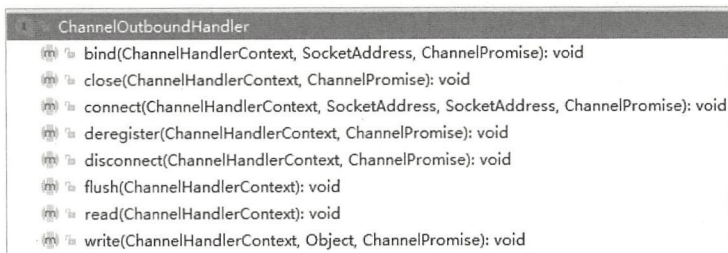


图 7-10 `ChannelOutboundHandler` 接口定义

如果用户需要同时处理 `inbound` 和 `outbound` 事件，可以同时实现上述两个接口，或者继承 `ChannelDuplexHandler` 类。需要指出的是，尽管 `ChannelHandler` 的线程安全性由使用方式决定，但是 `ChannelPipeline` 却是线程安全的，它可以被多个 `NioEventLoop` 线程并发调用，也可以在运行过程中动态地删除和添加 `ChannelHandler`，实现动态编排。例如，当业务高峰期需要对系统做拥塞保护时，就可以根据当前的系统时间进行判断，如果处于业务高峰期，则动态地将系统拥塞保护 `ChannelHandler` 添加到当前的 `ChannelPipeline` 中，当高峰期过去之后，就可以动态删除拥塞保护 `ChannelHandler` 了。

在添加或者删除 `ChannelHandler` 时，可能不同 `ChannelHandler` 对应不同的 `NioEventLoop` 执行，此时需要通过加锁的方式保障 `ChannelPipeline` 的并发安全。以添加 `ChannelHandler`





代码为例，通过 `synchronized` 块对 `ChannelPipeline` 实例加锁，完成链表的更新操作，代码如下（`DefaultChannelPipeline` 类）。如果 `ChannelPipeline` 没有多线程并发访问，则利用 JDK1.6 以上版本提供的锁消除机制，JVM 可以自动优化这把锁，不会影响系统性能。

---

```

public final ChannelPipeline addLast(EventExecutorGroup group, String name,
ChannelHandler handler) {
    final AbstractChannelHandlerContext newCtx;
    synchronized (this) {
        checkMultiplicity(handler);
        newCtx = newContext(group, filterName(name, handler), handler);
        addLast0(newCtx);
    }
    //后续代码省略
}

```

---

`ChannelPipeline` 通过链表的方式管理 `ChannelHandler`，每个 `ChannelHandler` 都对应一个上下文 `ChannelHandlerContext`，当新的 `ChannelHandler` 被添加到 `ChannelPipeline`，就会更新 `ChannelHandlerContext` 的位置指针，如图 7-11 所示。

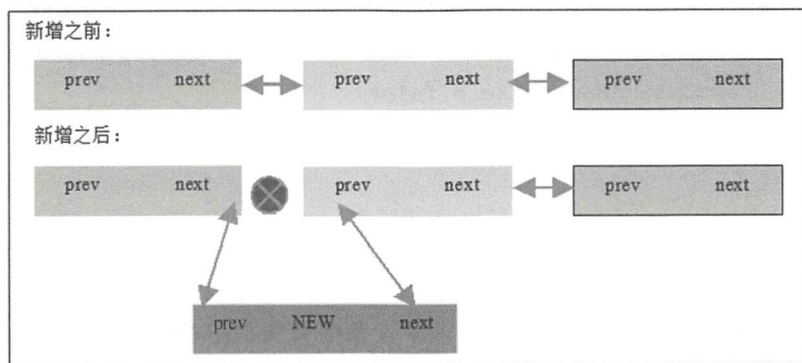


图 7-11 新增 `ChannelHandler` 时对应的 `ChannelHandlerContext` 位置指针变化

在实际业务场景中，`ChannelPipeline` 通常需要添加如下几类 `ChannelHandler`。

- (1) 协议解码 `ChannelHandler`。
- (2) 协议编码 `ChannelHandler`。
- (3) 业务逻辑执行 `ChannelHandler`。



对于耗时的业务逻辑执行，例如访问数据库、中间件、第三方系统等，则需要切换到业务线程池中，避免阻塞 Netty 的 `NioEventLoop` 线程，导致消息无法接收和发送。

## 7.2.2 用户自定义 Event 原理和源码分析

除了一些标准的网络 I/O 操作，例如链路创建、链路关闭、消息接收和发送等，还有一些非通用事件，通过 `UserEvent` 的方式触发和发送，比较典型的如 SSL/TLS 握手结果，当握手成功之后，触发 `SslHandshakeCompletionEvent.SUCCESS` 事件，业务通过监听该事件可以在 SSL/TLS 握手成功之后做特定的业务逻辑操作，示例代码如下（`SslHandler` 类）：

---

```
private void setHandshakeSuccess() {
    handshakePromise.trySuccess(ctx.channel());
    if (logger.isDebugEnabled()) {
        logger.debug("{} HANDSHAKEN: {}", ctx.channel(),
engine.getSession().getCipherSuite());
    }
    ctx.fireUserEventTriggered(SslHandshakeCompletionEvent.SUCCESS);
    if (readDuringHandshake && !ctx.channel().config().isAutoRead()) {
        readDuringHandshake = false;
        ctx.read();
    }
}
}
```

---

除了 SSL 握手结果通知，Netty 还提供了大量内置的 `UserEvent`，如图 7-12 所示。

熟悉这些系统的 `UserEvent`，可以在业务代码中监听并处理这些事件，实现业务功能的扩展和定制。例如，通过监听 SSL/TLS 握手成功事件可以对并发连接数做流控。除了 Netty 默认提供的 `UserEvent`，用户也可以根据业务需要定义并发送 `UserEvent`，其他的业务 `ChannelHandler` 通过监听该事件实现双方的解耦，例如，性能统计 `ChannelHandler A` 发布一个性能统计数据事件，包含 QPS、平均时延、成功率等数据，告警 `ChannelHandler` 通过监听性能统计事件，实现成功率低的业务告警，或者打印异常日志。通过事件订阅发布的机制可以实现业务逻辑的解耦。

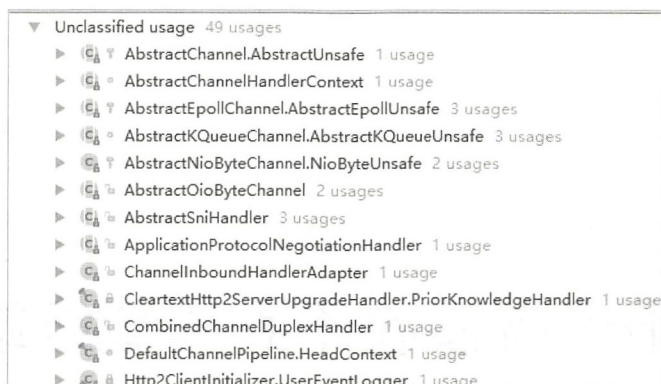


图 7-12 Netty 提供的 UserEvent 相关类库

通过调用 `ChannelHandlerContext.fireUserEventTriggered()` 或者 `Channel.pipeline().fireUserEventTriggered()` 可以实现业务自定义事件的发送。

## 7.3 总结

`ChannelHandler` 是用户最常用的接口，掌握了 `ChannelHandler` 及 `ChannelPipeline` 工作原理，就清楚了什么时候该使用共享的 `ChannelHandler`，什么时候该对 `ChannelHandler` 做并发保护。无论缺少保护还是过度保护，都会给业务带来副作用，甚至严重的功能或性能问题，因此 `ChannelHandler` 的并发安全性是非常重要的。

## 第 8 章

---

# 车联网服务端接收不到车载终端消息 案例

得益于高性能、低时延的优势，Netty 被广泛应用于物联网领域，用于海量终端设备的协议接入、消息收发和数据处理。

当服务端出现性能瓶颈或者阻塞时，就会导致终端设备连接超时和掉线，引发各种问题，因此在物联网场景下，一定要防止服务端因为编码不当导致的意外阻塞，进而无法处理终端请求消息。

### 8.1 车联网服务端接收不到车载终端消息问题

---

车联网服务端使用 Netty 构建，接收车载终端的请求消息，然后下发给后端其他系统，最后返回应答给车载终端。系统运行一段时间后发现服务端接收不到车载终端消息，导致业务中断，需要尽快找到产生问题的原因。

### 8.1.1 故障现象

服务端运行一段时间之后,发现无法接收到车载终端的消息,相关日志如图 8-1 所示。

Sat Aug 18 11:03:10 CST 2018→ Server receive client message : 4199
Sat Aug 18 11:03:10 CST 2018→ Server receive client message : 4200
Sat Aug 18 11:03:25 CST 2018→ Server receive client message : 4201
Sat Aug 18 11:03:25 CST 2018→ Server receive client message : 4202
Sat Aug 18 11:03:25 CST 2018→ Server receive client message : 4203
Sat Aug 18 11:03:25 CST 2018→ Server receive client message : 4204

图 8-1 车联网服务端无法接收消息的相关日志

从日志看,服务端每隔一段时间(示例中是 15s,实际业务时间是随机的)就会接收不到消息,隔一段时间之后恢复,然后又没消息,周而复始。跟车载终端确认,终端设备每隔固定时间就会发送消息给服务端(日志分析),因此排除是因为终端没发消息导致出现问题的可能。

接着怀疑是不是服务端负载过重,抢占不到 CPU 资源导致的周期性阻塞。采集 CPU 使用情况数据,发现 CPU 资源不是瓶颈,排除因为 CPU 资源导致出现问题的可能,如图 8-2 所示。

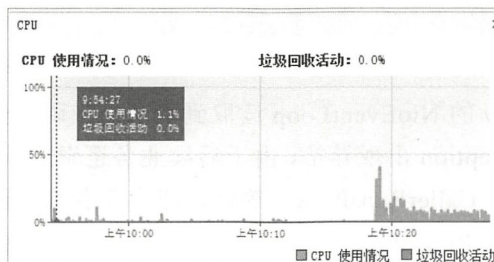


图 8-2 车联网服务端 CPU 使用情况

CPU 不是瓶颈,是不是内存有问题,导致频繁 GC 引起业务线程暂停?采集 GC 统计数据,如图 8-3 所示。

▼ GC 表			
PS Scavenge PS MarkSweep			
名称	值	类型	更新间隔
Total Collection Time	65 ms	持续时间	默认值
Collection Count	24	数值	默认值
GC Start Time	39 min 49 s	持续时间	默认值
GC End Time	39 min 49 s	持续时间	默认值
GC Duration	2 ms	持续时间	默认值
GC ID	24	数值	默认值
GC Thread Count	4	数值	默认值

图 8-3 车联网服务端 GC 统计数据

通过 CPU 和内存资源占用监控分析，发现硬件资源不是瓶颈，问题应该出在服务端。

### 8.1.2 故障期线程堆栈快照分析

从现象上看，服务端接收不到消息，排除 GC、网络等问题之后，很有可能是 Netty 的 `NioEventLoop` 线程阻塞，导致 TCP 缓冲区的数据没有及时读取，故障期间采集服务端的线程堆栈进行分析，如图 8-4 所示。

```
"nioEventLoopGroup-3-1" #13 prio=10 os_prio=2 tid=0x0000000017373800 nid=0x2984 waiting on condition [0x0000000017e3e000]
  java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:340)
    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
    at io.netty.cases.chapter.demo8.IotCarsServerHandler.lambda$channelRead$0(IotCarsServerHandler.java:47)
    at io.netty.cases.chapter.demo8.IotCarsServerHandler$$Lambda$2/230063144.run(Unknown Source)
    at java.util.concurrent.ThreadPoolExecutor$CallerRunsPolicy.rejectedExecution(ThreadPoolExecutor.java:2022)
    at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:823)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1369)
    at io.netty.cases.chapter.demo8.IotCarsServerHandler.channelRead(IotCarsServerHandler.java:39)
    at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:362)
```

图 8-4 故障期间服务端的线程堆栈

从线程堆栈看，Netty 的 `NioEventLoop` 读取到消息后，调用业务线程池执行业务逻辑时，`RejectedExecutionException` 出现异常，由于后续业务逻辑由 `NioEventLoop` 线程执行，因此可以判断业务使用了 `CallerRunsPolicy` 策略，即在业务线程池消息队列满之后，由调用方的线程来执行当前的 `Runnable`。`NioEventLoop` 在执行业务任务时发生了阻塞，导致 `NioEventLoop` 线程无法处理网络读写消息，因此会看到服务端没有消息接入，从阻塞状态恢复之后，就可以继续接收消息。

对源码（为了便于理解，作为示例模拟的故障代码）进行分析，很快发现了问题（`IotCarsServerHandler` 类）所在：

```
public class IotCarsServerHandler extends ChannelInboundHandlerAdapter {
    static AtomicInteger sum = new AtomicInteger(0);

    static ExecutorService executorService = new ThreadPoolExecutor(1, 3, 30,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(1000), new ThreadPoolExecutor.
```



```

CallerRunsPolicy());
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println(new Date() + "--> Server receive client message :
" + sum.incrementAndGet());
        executorService.execute(()->
        {
            ByteBuf req = (ByteBuf) msg;
            //其他业务逻辑处理, 访问数据库
            if (sum.get() % 100 == 0 || (Thread.currentThread()==
ctx.channel().eventLoop()))
                try
                {
                    //访问数据库, 模拟偶现的数据库慢, 同步阻塞 15s
                    TimeUnit.SECONDS.sleep(15);
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            //转发消息, 此处代码省略, 转发成功之后返回响应给终端
            ctx.writeAndFlush(req);
        });
    }
    //后续代码省略
}

```

如果后端业务逻辑处理慢, 则会导致业务线程池阻塞队列积压, 当积压达到容量上限时, JDK 会抛出 `RejectedExecutionException` 异常, 由于业务设置了 `CallerRunsPolicy` 策略, 就会由调用方的线程 `NioEventLoop` 执行业务逻辑, 最终导致 `NioEventLoop` 线程被阻塞, 无法读取请求消息。

除了 JDK 线程池异常处理策略使用不当, 有些业务人员喜欢自己写阻塞队列, 当队列满时, 向队列加入新的消息会阻塞当前线程, 直到消息能够加入队列。案例中的车联网服务端真实业务代码就有此类问题: 当转发到下游系统发生某些故障时, 会导致业务定义

的阻塞队列无法弹出消息进行处理，当队列积压满时，就会阻塞 Netty 的 NIO 线程，而且无法自动恢复。

当系统拥塞时，可以采用丢弃当前消息或者流控的方式避免问题进一步恶化，对上述案例代码做修改，当队列满时采用丢弃策略，防止阻塞 Netty 的 `NioEventLoop` 线程（`IotCarsDiscardServerHandler` 类）：

---

```
public class IotCarsDiscardServerHandler extends ChannelInboundHandlerAdapter {
    static AtomicInteger sum = new AtomicInteger(0);
    static ExecutorService executorService = new ThreadPoolExecutor(1, 3, 30,
        TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(1000), new ThreadPoolExecutor.
DiscardPolicy());
    //后续代码省略
}
```

---

### 8.1.3 `NioEventLoop` 线程防挂死策略

由于 `ChannelHandler` 是业务代码和 Netty 框架交汇的地方，`ChannelHandler` 里面的业务逻辑通常由 `NioEventLoop` 线程执行，因此防止业务代码阻塞 `NioEventLoop` 线程就显得非常重要，常见的阻塞情况有两类。

（1）直接在 `ChannelHandler` 中写可能导致程序阻塞的代码，包括但不限于数据库操作、第三方服务调用、中间件服务调用、同步获取锁、`Sleep` 等。

（2）切换到业务线程池或者业务消息队列做异步处理时发生了阻塞，最典型的有阻塞队列、同步获取锁等。

在实际项目中，推荐业务处理线程和 Netty 网络 I/O 线程分离策略，原因如下。

（1）充分利用多核的并行处理能力：I/O 线程和业务线程分离，双方可以并行处理网络 I/O 和业务逻辑，充分利用多核的并行计算能力，提升性能。

（2）故障隔离：后端的业务线程池处理各种类型的业务消息，有些是 I/O 密集型的、有些是 CPU 密集型的、有些是纯内存计算型的，不同的业务处理时延，以及发生故障的

概率都是不同的。如果把业务线程和 I/O 线程合并，就会存在如下问题。

① 某类业务处理较慢，阻塞 I/O 线程，导致其他处理较快的业务消息的响应无法及时发送出去。

② 即便同类业务，使用同一个 I/O 线程同时处理业务逻辑和 I/O 读写，如果请求消息的业务逻辑处理较慢，同样会导致响应消息无法及时发送出去。

(3) 可维护性：I/O 线程和业务线程分离之后，双方职责单一，有利于代码维护和问题定位。如果合并在一起执行，当 RPC 调用时延增大时，到底是网络问题、I/O 线程问题还是业务逻辑问题导致的时延大，问题定位难度非常大。例如，在业务线程中访问缓存或者数据库偶尔时延增大，就会导致 I/O 线程被阻塞，时延出现毛刺，这些时延毛刺的定位难度非常大。

Netty I/O 线程和业务逻辑处理线程分离之后的线程模型如图 8-5 所示。

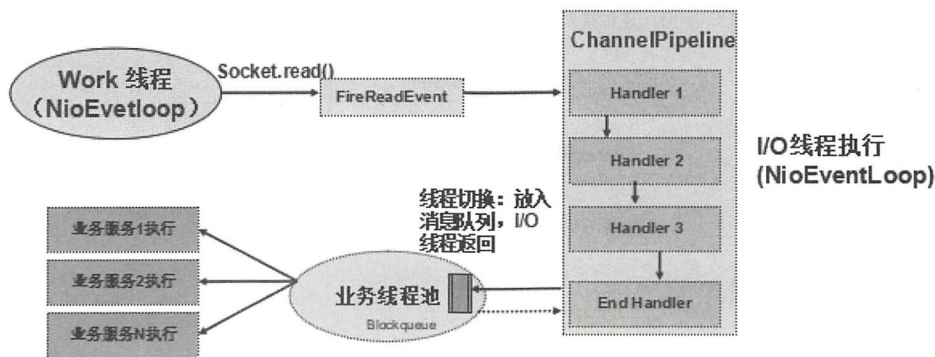


图 8-5 Netty I/O 线程和业务逻辑处理线程分离之后的线程模型

## 8.2 NioEventLoop 线程工作机制

Netty 的 NioEventLoop 并不是一个纯粹的 I/O 线程，它除了负责 I/O 读写操作，还兼顾以下两类任务。

(1) 系统任务：通过调用 NioEventLoop 的 execute(Runnable task)方法执行，Netty 有很多系统任务，创建它们的主要原因是，当 I/O 线程和用户线程同时操作网络资源时，为

了防止并发操作导致的锁竞争，将用户线程的操作封装成任务放入消息队列，由 I/O 线程负责执行，这样就实现了局部无锁化。

(2) 定时任务：通过调用 `NioEventLoop` 的 `schedule(Runnable command, long delay, TimeUnit unit)` 系列方法实现。

## 8.2.1 I/O 读写操作原理和源码分析

`NioEventLoop` 作为 `Reactor` 线程，负责 TCP 连接的创建和接入，以及 TCP 消息的读写，`Reactor` 服务端线程模型如图 8-6 所示。

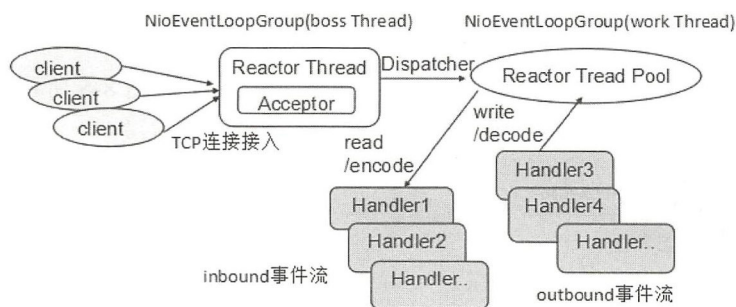


图 8-6 Reactor 服务端线程模型

`Reactor` 线程的职责如下。

- (1) 作为 NIO 服务端，接收客户端的 TCP 连接。
- (2) 作为 NIO 客户端，向服务端发起 TCP 连接。
- (3) 读取通信对端请求或者应答消息。
- (4) 向通信对端发送消息请求或者应答消息。

由于 `Reactor` 模式使用的是异步非阻塞 I/O，因此所有的 I/O 操作都不会导致阻塞，理论上一个线程可以独立处理所有 I/O 相关的操作。但是这对于高负载、大并发的应用场景却不合适，原因如下。

(1) 一个 NIO 线程同时处理成百上千条链路，在性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的编码、解码、读取和发送需求。

(2) 当 NIO 线程负载过重时, 处理速度将变慢, 这会导致大量客户端连接超时, 超时之后往往会进行重发消息, 这加重了 NIO 线程的负载, 最终会导致大量消息积压和处理超时, 成为系统的性能瓶颈。

(3) 可靠性问题: 一旦 NIO 线程意外跑飞, 或者进入死循环, 会导致整个系统通信模块不可用, 不能接收和处理外部消息, 造成节点故障。

对于 Netty, 在创建 `NioEventLoopGroup` 时可以指定工作的 I/O 线程数, 通常为“CPU 内核数 $\times$ 2”或者“CPU 内核数+1”, 这样可提升网络的读写性能。需要指出的是, 不要把 I/O 线程数设置得过大, 除了会导致线程竞争加剧, 还会带来其他副作用。

`NioEventLoop` 线程处理网络读写等操作的关键是聚合了一个 `Selector`, 代码如下:

---

```
public final class NioEventLoop extends SingleThreadEventLoop {
    {
        private Selector selector;

        private SelectorTuple openSelector() {
            final Selector unwrappedSelector;
            try {
                unwrappedSelector = provider.openSelector();
            } catch (IOException e) {
                throw new ChannelException("failed to open a new selector", e);
            }

            if (DISABLE_KEYSET_OPTIMIZATION) {
                return new SelectorTuple(unwrappedSelector);
            }

            //后续代码省略
        }
    }
}
```

---

除了支持 JDK 原生的 `Selector`, Netty 也支持创建其他 SPI 提供的 `Selector`, 同时 Netty 对 `Selector` 的遍历也做了性能优化。对于网络消息的处理, 通过轮询 `Selector` 的 `SelectedSelectionKeySet` 实现, 代码如下 (`NioEventLoop` 类):

---

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
```

---



```
final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
if (!k.isValid()) {
    final EventLoop eventLoop;
    //代码省略
try {
    int readyOps = k.readyOps();
    if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
        int ops = k.interestOps();
        ops &= ~SelectionKey.OP_CONNECT;
        k.interestOps(ops);
        unsafe.finishConnect();
    }
    if ((readyOps & SelectionKey.OP_WRITE) != 0) {
        ch.unsafe().forceFlush();
    }
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) !=
0 || readyOps == 0) {
        unsafe.read();
    }
} catch (CancelledKeyException ignored) {
    unsafe.close(unsafe.voidPromise());
}
}
```

---

通过对 SelectionKey 的取值进行判断，完成对应的 I/O 操作。

- (1) 如果为 OP\_CONNECT，则代表客户端异步连接操作执行结果。
- (2) 如果为 OP\_WRITE，说明发生了写半包，发送队列尚有消息未完成发送，需要继续执行发送操作。
- (3) 如果为 OP\_READ，说明 SocketChannel 上有消息可以读取，执行 read ByteBuffer 操作。

(4) 如果为 OP\_ACCEPT, 说明 ServerSocketChannel 上有新的客户端 TCP 连接接入, 需要执行 accept 操作, 完成 TCP 握手和客户端 TCP 连接的接入。

## 8.2.2 异步任务执行原理和源码分析

除了一些标准的网络 I/O 操作, NioEventLoop 也支持各种 Runnable 类型的任务的执行, 任务的使用有两种场景。

(1) Netty 系统任务, 主要用于任务的异步执行, 或者用于从用户线程切换到 Netty 的 NioEventLoop 线程, 避免业务 ChannelHandler 加锁。

(2) 用户自定义用来辅助 I/O 操作的业务任务。

AbstractWriteTask 就是比较典型的 Netty 系统任务, 它将 write 操作封装成任务, 放入 NioEventLoop 任务队列异步执行, 代码如下 (AbstractWriteTask 类):

---

```
abstract static class AbstractWriteTask implements Runnable {
    //后续代码省略...
    public final void run() {
        try {
            if (ESTIMATE_TASK_SIZE_ON_SUBMIT) {
                ctx.pipeline.decrementPendingOutboundBytes(size);
            }
            write(ctx, msg, promise);
        } finally {
            ctx = null;
            msg = null;
            promise = null;
            handle.recycle(this);
        }
    }
    //后续代码省略
}
```

---

任务存放在 `SingleThreadEventExecutor` 类的成员变量 `Queue<Runnable> taskQueue` 中，每次 `Selector` 轮询完，执行 `taskQueue` 中的任务，代码如下（`SingleThreadEventExecutor` 类）：

---

```
protected boolean runAllTasks() {
    assert inEventLoop();
    boolean fetchedAll;
    boolean ranAtLeastOne = false;
    do {
        fetchedAll = fetchFromScheduledTaskQueue();
        if (runAllTasksFrom(taskQueue)) {
            ranAtLeastOne = true;
        }
    } while (!fetchedAll);
    if (ranAtLeastOne) {
        lastExecutionTime = ScheduledFutureTask.nanoTime();
    }
    afterRunningAllTasks();
    return ranAtLeastOne;
}
}
```

---

由于 `NioEventLoop` 需要同时处理 I/O 事件和非 I/O 任务，为了保证两者都能得到足够的 CPU 时间，Netty 提供了 I/O 比例供用户定制。如果 I/O 操作多于定时任务和其他任务，则可以将 I/O 比例调大，反之则调小，默认值为 50%，相关代码如下（`NioEventLoop` 类）：

---

```
protected void run() {
    //代码省略
    final int ioRatio = this.ioRatio;
    if (ioRatio == 100) {
        try {
            processSelectedKeys();
        } finally {
            runAllTasks();
        }
    }
}
```

---

```

    }
} else {
    final long ioStartTime = System.nanoTime();
    try {
        processSelectedKeys();
    } finally {
        final long ioTime = System.nanoTime() - ioStartTime;
        runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
    }
}
//代码省略
}

```

当限制的执行时间到期时，无论当前积压的任务是否执行完，都需要退出循环，防止长时间执行任务而阻塞网络 I/O 操作。

### 8.2.3 定时任务执行原理和源码分析

除了不同的 `Runnable` 类型的任务，`NioEventLoop` 还支持执行定时任务，通过调用 `schedule` 接口，可以实现定时任务的执行，相关接口定义如图 8-7 所示（`AbstractScheduledEventExecutor` 类）。

```

m  schedule(Callable<V>, long, TimeUnit): ScheduledFuture<V> |AbstractEventExecutor
m  schedule(Runnable, long, TimeUnit): ScheduledFuture<?> |AbstractEventExecutor
m  schedule(ScheduledFutureTask<V>): ScheduledFuture<V>
m  scheduleAtFixedRate(Runnable, long, long, TimeUnit): ScheduledFuture<?> |AbstractEventExecutor
m  scheduledTaskQueue(): PriorityQueue<ScheduledFutureTask<?>>
m  scheduleWithFixedDelay(Runnable, long, long, TimeUnit): ScheduledFuture<?> |AbstractEventExecutor

```

图 8-7 `NioEventLoop` 执行定时任务相关接口定义

通过调用 `fetchFromScheduledTaskQueue`，将到期的定时任务加入 `taskQueue` 并随 `taskQueue` 执行，实际上可以理解为 `taskQueue` 本身就是需要立即执行的定时任务队列，相关代码如下（`SingleThreadEventExecutor` 类）：

```

private boolean fetchFromScheduledTaskQueue() {
    long nanoTime = AbstractScheduledEventExecutor.nanoTime();

```

```

        Runnable scheduledTask = pollScheduledTask(nanoTime);
        while (scheduledTask != null) {
            if (!taskQueue.offer(scheduledTask)) {
                scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);
                return false;
            }
            scheduledTask = pollScheduledTask(nanoTime);
        }
        return true;
    }
}

```

---

在 Netty 中，定时任务最经典的使用场景就是链路空闲状态检测，在初始化 `IdleStateHandler` 时，同步创建 `ReaderIdleTimeoutTask`、`WriterIdleTimeoutTask` 和 `AllIdleTimeoutTask` 三个定时任务，负责链路空闲状态检测，相关代码如下：

```

private void initialize(ChannelHandlerContext ctx) {
    //代码省略
    lastReadTime = lastWriteTime = ticksInNanos();
    if (readerIdleTimeNanos > 0) {
        readerIdleTimeout = schedule(ctx, new ReaderIdleTimeoutTask(ctx),
            readerIdleTimeNanos, TimeUnit.NANOSECONDS);
    }
    if (writerIdleTimeNanos > 0) {
        writerIdleTimeout = schedule(ctx, new WriterIdleTimeoutTask(ctx),
            writerIdleTimeNanos, TimeUnit.NANOSECONDS);
    }
    if (allIdleTimeNanos > 0) {
        allIdleTimeout = schedule(ctx, new AllIdleTimeoutTask(ctx),
            allIdleTimeNanos, TimeUnit.NANOSECONDS);
    }
}
}

```

---

对于用户而言，如果需要执行一些周期性的任务，不需要自己创建定时器或者使用 JDK 的 `ScheduledExecutorService`，可以直接使用 Netty 的 `NioEventLoop` 定时任务，实现



诸如心跳发送等功能。

### 8.2.4 Netty 多线程最佳实践

Netty 的多线程编程最佳实践如下。

(1) 创建两个 `NioEventLoopGroup`，用于逻辑隔离 NIO Acceptor 和 NIO I/O 线程。

(2) 尽量不要在 `ChannelHandler` 中启动用户线程（解码后用于将 POJO 消息派发到后端业务线程的除外）。

(3) 解码要放在 NIO 线程调用的解码 Handler 中进行，不要切换到用户线程完成消息的解码。

(4) 如果业务逻辑操作非常简单（纯内存操作），没有复杂的业务逻辑计算，也没有可能会导致线程被阻塞的磁盘操作、数据库操作、网络操作等，可以直接在 NIO 线程上完成业务逻辑编排，不需要切换到用户线程。

(5) 如果业务逻辑复杂，不要在 NIO 线程上完成，建议将解码后的 POJO 消息封装成任务，派发到业务线程池中由业务线程执行，以保证 NIO 线程尽快被释放，处理其他的 I/O 操作。

推荐的线程数量计算公式有以下两种。

(1) 公式 1：线程数量 = (线程总时间 / 瓶颈资源时间) × 瓶颈资源的线程并行数。

(2) 公式 2：QPS = 1000 / 线程总时间 × 线程数。

由于用户场景不同，对于一些复杂的系统，实际上很难计算出最优线程配置，只能根据测试数据和用户场景，结合公式给出一个相对合理的范围，然后对范围内的数据进行性能测试，选择相对最优值。

## 8.3 总结

当 Netty 服务端接收不到消息时，首先需要检查是客户端没有发送到服务端，还是服

务端没有读取消息。导致服务端无法读取消息的原因有很多，常见的包括 GC 导致的应用线程暂停、服务端的 `NioEventLoop` 线程被意外阻塞等。通过网络 I/O 线程和业务逻辑线程分离，可以实现双方的并行处理，提升系统的可靠性。对于用户而言，在编写代码时，始终需要考虑 `NioEventLoop` 线程是否会被业务代码阻塞，只有消除所有可能导致的阻塞点，才能保证程序稳定运行。

## 第 9 章

---

# Netty 3.X 版本升级案例

目前 Netty 的主流版本有 3 个，分别是 3.X、4.0.X 和 4.1.X，绝大多数新业务都会采用 4.1.X 或者 4.0.X 进行开发，对于已经使用 3.X 开发的遗留系统而言，是否主动升级 Netty 版本是一个两难选择：如果不升级，一些新版本的特性，例如 HTTP/2、MQTT、内存池等将无法使用；如果升级，由于包路径、很多 API 接口类库都需要重构，修改工作量大，稍有不慎就会踩坑。

包路径、API 的修改是显式修改，升级之后代码编译就会报错，这类问题相对好解决，但是底层的内存分配策略、线程模型的修改却是隐式的，如果对两个版本的架构差异理解不透彻，就会遭遇性能下降、内存泄漏等诸多问题。本章对从 Netty 3.X 升级到 Netty 4.X 遇到的典型问题进行总结和分析，希望对需要进行版本升级的读者有帮助。

## 9.1 Netty 3.X 的版本升级背景

---

相比于其他开源项目，Netty 用户的版本升级之路更加艰辛，最根本的原因就是 Netty 4.X 对 Netty 3.X 没有做到很好的前向兼容。由于版本不兼容，大多数老版本使用者的想法就是，既然升级这么麻烦，暂时又不需要使用 Netty 4.X 的新特性，当前版本还挺稳定，

就暂时先不升级，以后看看再说。

坚守老版本还有很多其他的理由，例如线上系统的稳定性、对新版本的熟悉程度等。无论如何，升级 Netty 都是一件大事，特别是对 Netty 有直接强依赖的产品。从上面的分析可以看出，坚守老版本似乎是个不错的选择。但是，“理想是美好的，现实却是残酷的”，坚守老版本并非总是那么容易的，有时候可能会被迫升级。

### 9.1.1 被迫升级场景

除了为使用新特性而主动进行的版本升级，大多数升级都是“被迫的”，下面对这些升级原因进行总结。

(1) 公司的开源软件管理策略：对于那些大公司，不同部门和产品线依赖的开源软件版本经常不同，为了对开源依赖进行统一管理，降低安全、维护和管理成本，往往会指定优选的软件版本。由于 Netty 4.X 已经非常成熟，所以很多公司都优选 Netty 4.X。

(2) 维护成本：无论是依赖 Netty 3.X，还是 Netty 4.X，往往都需要在原框架之上做定制。例如客户端的断连重连、心跳检测、流控等。分别对 Netty 4.X 和 3.X 实现两套定制框架，开发和维护成本都非常高。根据开源软件的使用策略，当存在版本冲突的时候，往往会选择升级到更高的版本。Netty 依然遵循这个规则。

(3) 新特性：Netty 4.X 相比 Netty 3.X，提供了很多新的功能，例如优化的内存池、对 MQTT 和 HTTP/2 的支持等。如果用户需要使用这些新特性，最简便的做法就是将 Netty 升级到 4.X，而不是在 Netty 3.X 的基础上自己开发一套实现。

(4) 更优异的性能：Netty 4.X 相比 3.X，优化了内存池，减少了 GC 的频率，降低了内存消耗；通过优化 Reactor 线程池模型，用户的开发更加简单，线程调度也更加高效。

### 9.1.2 升级不当遭遇各种问题

从表面上看，类库包路径的修改、API 的重构等似乎是升级的重头戏，大家往往把注意力放到这些“明枪”上，但真正致命的却是“暗箭”。如果对 Netty 底层的事件调度机制和线程模型不熟悉，往往就会“中箭”。

后面几节以比较典型的真实问题为例，通过问题描述、定位和总结，让这些隐藏的“暗箭”充分暴露出来。

由于 Netty 4.X 线程模型改变导致的升级事故很多，限于篇幅，不一一枚举，这些问题万变不离其宗，只要抓住线程模型这个关键点，所有的疑难问题都能迎刃而解。

## 9.2 版本升级后数据被篡改问题

某业务产品，将 Netty3.X 升级到 4.X 之后，在系统运行过程中，偶现服务端发送给客户端的应答数据被莫名“篡改”。

业务服务端的处理流程如下。

- (1) 将解码后的业务消息封装成任务，投递到后端的业务线程池执行。
- (2) 业务线程处理业务逻辑，完成后构造应答消息发送给客户端。
- (3) 业务应答消息的编码通过继承 Netty 的 CodeC 框架实现，即 Encoder ChannelHandler。
- (4) 调用 Netty 的消息发送接口后，流程继续，根据业务场景，可能会继续操作原来发送的业务对象。

业务流程的代码片段示例如下：

---

```
//代码省略
//构造订购应答消息
SubInfoResp infoResp = new SubInfoResp();
//根据业务逻辑，对应答消息赋值
infoResp.setResultCode(0);
infoResp.setXXX();
//后续赋值操作省略
//调用 ChannelHandlerContext 进行消息发送
ctx.writeAndFlush(infoResp);
//消息发送完成，后续根据业务流程进行分支处理，修改 infoResp 对象
```



```
infoResp.setXXX();  
//后续代码省略
```

## 9.2.1 数据篡改原因分析

首先对应答消息被非法篡改的原因进行分析，经过定位发现被篡改的内容是调用 `writeAndFlush` 接口后，由后续业务分支代码修改应答消息导致的。由于修改操作在 `writeAndFlush` 操作之后，按照 Netty 3.X 的线程模型不应该出现该问题。

在 Netty3.X 中，`downstream` 是在业务线程里执行的，也就是说对 `SubInfoResp` 的编码操作是在业务线程中执行的，当编码后的 `ByteBuf` 对象被投递到消息发送队列时，业务线程才会返回并继续执行后续的业务逻辑，此时修改应答消息是不会改变已完成编码的 `ByteBuf` 对象的，所以肯定不会出现应答消息被篡改的问题。

经初步分析，应该是由于线程模型发生变更导致的问题，随后查验了 Netty 4.X 的线程模型，果然发生了变化：当调用 `write` 方法向外发送消息的时候，Netty 会将发送事件封装成任务，投递到 `NioEventLoop` 的任务队列异步执行，然后返回业务调用线程，业务线程继续执行，如果在响应对象编码之前业务线程和 Netty 的 `NioEventLoop` 线程同时修改了 `infoResp` 对象，就会发生并发问题。如果业务线程先执行了修改操作，则修改操作会影响后续编码的结果，造成数据被非法篡改。由于线程的执行先后顺序无法预测，因此该问题隐藏得相当深。如果不了解 Netty 两个版本的线程模型，就会掉入陷阱。

Netty 3.X 的业务逻辑没有问题，它的消息发送流程如图 9-1 所示。

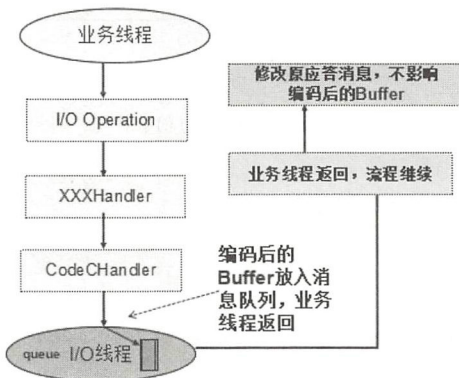


图 9-1 Netty 3.X 消息发送流程

升级到 Netty 4.X 之后，业务流程由于 Netty 线程模型的变更而发生改变，导致业务逻辑发生问题，如图 9-2 所示。

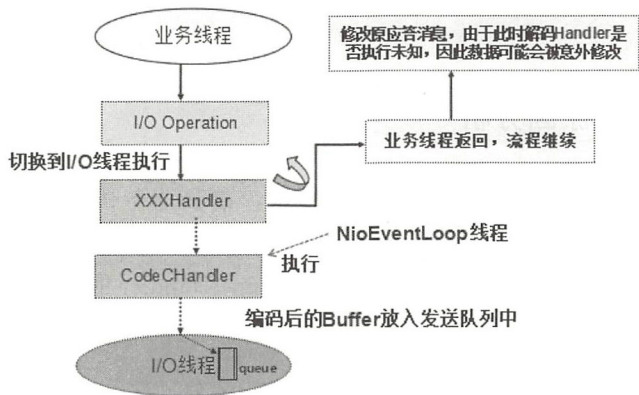


图 9-2 Netty 4.X 消息发送流程

## 9.2.2 问题总结

很多用户在进行 Netty 版本升级的时候，只关注包路径、类库和 API 的变更，并没有注意到隐藏在背后的“暗箭”——线程模型变更。

升级到 Netty 4.X 的用户需要根据新的线程模型对已有的系统进行评估，重点需要关注 outbound 的 ChannelHandler，如果它的正确性依赖于 Netty 3.X 的线程模型，则很可能在新的线程模型中出现问题，也许是功能问题，也许是性能等其他问题。

## 9.3 升级后上下文丢失问题

为了提升业务的二次定制能力，降低对接口的侵入性，业务使用线程变量进行消息上下文的传递，例如消息发送源地址信息、消息 ID、会话 ID 等。业务同时使用了一些第三方开源容器，也提供了线程级变量上下文的功能。业务通过容器上下文获取第三方容器的系统变量信息。

升级到 Netty 4.X 之后，业务继承自 Netty 的 `ChannelHandler` 发生了空指针异常，无论是业务自定义的线程上下文，还是第三方容器的线程上下文，都取不到传递的变量值，但是在 Netty 3.X 中却没有问题。

### 9.3.1 上下文丢失原因分析

---

首先检查代码，看业务是否传递了相关变量，确认业务传递了后再怀疑问题是否跟 Netty 版本升级相关，调试发现，业务 `ChannelHandler` 获取的线程上下文对象和之前业务传递的上下文对象不是同一个。这就说明执行 `ChannelHandler` 的线程跟处理业务的线程不是同一个线程。

查看 Netty 4.X 线程模型的 API 文档发现，Netty 修改了 outbound 的线程模型，正好影响了业务消息发送时的线程上下文传递，最终导致业务线程变量丢失。

### 9.3.2 依赖第三方线程模型的思考

---

通常业务的线程模型有如下几种。

- (1) 业务自定义线程池/线程组处理业务，例如使用 JDK 1.5 提供的 `ExecutorService`。
- (2) 使用 J2EE Web 容器自带的线程模型，常见的如 JBoss 和 Tomcat 的 HTTP 接入线程（如 HTTP Connector 线程池）等。
- (3) 隐式使用其他第三方框架的线程模型，例如使用 NIO 框架进行协议处理，除非业务明确地使用自定义线程模型，业务代码就隐式使用了 NIO 框架的线程模型。

在实践中我们发现很多业务使用了第三方框架，但是只熟悉 API 和功能，对线程模型并不清楚。某个类库由哪个线程调用，糊里糊涂。为了方便变量传递，又随意地使用线程变量，实际对背后第三方类库的线程模型产生了强依赖。在容器或者第三方类库升级之后，如果线程模型发生了变更，则原有功能就会出现问題。

鉴于此，在实际工作中，尽量不要强依赖第三方类库的线程模型，如果确实无法避免，则必须对它的线程模型有深入和清晰的了解。在第三方类库升级之后，需要检查线程模型是否发生了变更，如果发生了变更，相关的代码也需要同步升级。

## 9.4 升级后应用遭遇性能下降问题

根据 Netty 官方提供的性能测试数据,在同样的性能测试场景下,Netty 4.X 相比 Netty 3.X,GC 中断频率降低了 80%,垃圾生成速度则为原来的 20%左右,相比老版本,Netty 4.X 在内存的申请和分配上性能得到了极大提升。

正是因为看到了相关的性能测试对比报告,很多用户选择了将 Netty 版本升级到 4.X。事后一些用户反馈 Netty 4.X 并没有给产品带来预期的性能提升,有些甚至还出现了非常严重的性能下降,下面我们就以某业务产品的升级经历为案例,详细分析一下导致性能下降的原因。

### 9.4.1 性能下降原因分析

通过对热点方法的分析,发现在消息发送过程中,有两处热点。

(1) 消息发送性能统计相关 ChannelHandler。

(2) 返回响应时的编码 ChannelHandler (业务消息反序列化)。

对使用 Netty 3.X 的业务产品进行性能对比测试,发现上述两个 ChannelHandler 同样也是热点方法。既然都是热点方法,为什么切换到 Netty 4.X 之后性能会下降呢?

通过方法的调用堆栈分析发现了两个版本的差异:在 Netty 3.X 中,上述两个热点方法都是由业务线程负责执行的;而在 Netty 4.X 中,则是由 NioEventLoop(I/O)线程执行的。对于某个链路的 write 操作,业务侧拥有多个线程的线程池,而 NioEventLoop 线程只有一个,所以执行效率更低,返回客户端的应答时延就大。随着时延增大,会导致系统并发量降低,性能下降。

找出问题的根因后,针对 Netty 4.X 的线程模型对业务进行专项优化(在业务线程中将耗时的反序列化等逻辑操作完成,不在业务 ChannelHandler 中执行),性能达到预期,远超 Netty 3.X 的性能。

Netty 3.X 的消息发送线程模型如图 9-3 所示,充分利用了业务多线程并行编码和 ChannelHandler 处理的优势,在一个周期 T 内可以处理 N 条业务消息。



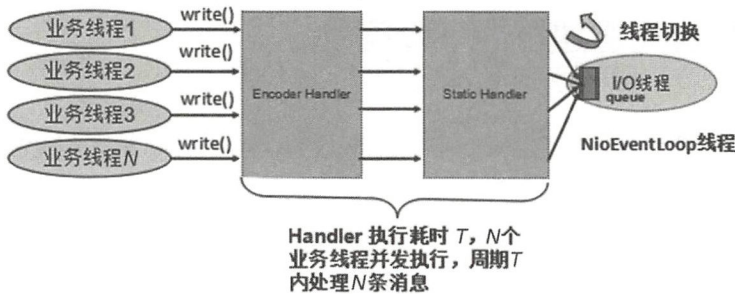


图 9-3 Netty 3.X 的消息发送线程模型

切换到 Netty 4.X 之后，业务耗时 ChannelHandler 被 I/O 线程串行执行，因此性能产生了比较大的下降，Netty 4.X 的消息发送线程模型如图 9-4 所示。

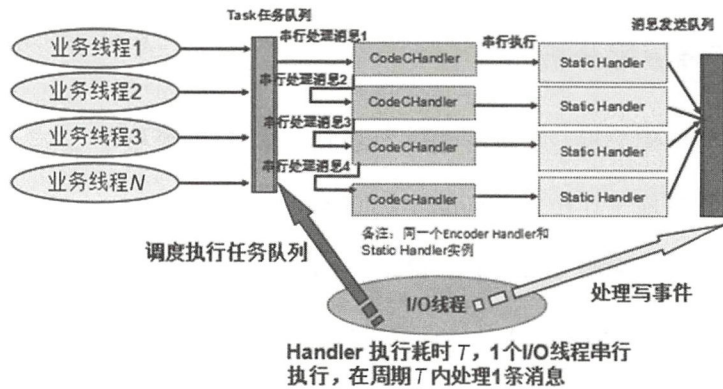


图 9-4 Netty 4.X 的消息发送线程模型

### 9.4.2 性能优化建议

适当地调大 work 线程组的线程数 (NioEventLoopGroup)，分担每个 NioEventLoop 线程的负载，提升 ChannelHandler 执行的并发度。同时，将业务上耗时的操作从 ChannelHandler 中移除，放入业务线程池处理，提升对热点代码的执行效率。对于不适合转移到业务线程处理的一些耗时逻辑，也可以通过为 ChannelHandler 绑定线程池的方式提升性能，如图 9-5 所示 (ChannelPipeline 接口)。



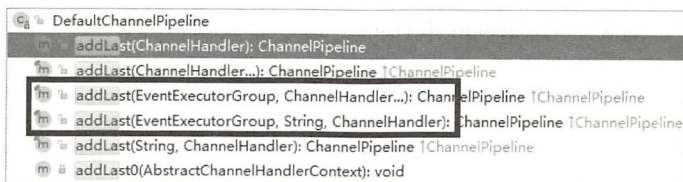


图 9-5 为业务 ChannelHandler 绑定线程池

## 9.5 Netty 线程模型变更分析

在上面几个案例中，都是因为不了解或者没注意 Netty 4.X 线程模型发生了变化，才踩了“坑”。本节对两个版本的线程模型进行分析，以便大家更好地了解和掌握变化点。

### 9.5.1 Netty 3.X 版本线程模型

Netty 3.X 的 I/O 操作分为两大类事件。

(1) Upstream ChannelEvent：主要包括链路建立事件、链路激活事件、读事件、I/O 异常事件、链路关闭事件等。

(2) Downstream ChannelEvent：主要包括写事件、连接事件、监听绑定事件等。

我们首先分析 Netty 3.X 的 Upstream ChannelEvent 相关线程模型，如图 9-6 所示。

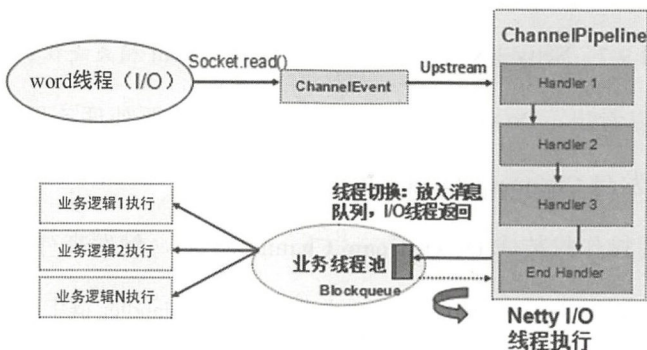


图 9-6 Netty 3.X 的 Upstream ChannelEvent 相关线程模型





从图 9-6 中可以看出，Upstream 事件的主要处理流程如下。

- (1) I/O 线程 (work 线程) 将消息从 TCP 缓冲区读取到 SocketChannel 的接收缓冲区。
- (2) 由 I/O 线程负责生成相应的事件 (例如 UpstreamMessageEvent)，触发事件向上执行，调度到 ChannelPipeline 中。
- (3) I/O 线程调度执行 ChannelPipeline 中 Handler 链的对应方法，直到业务实现的最后一个 Handler。
- (4) 最后一个 Handler 将消息封装成 Runnable，放入业务线程池执行，I/O 线程返回，继续执行读、写等 I/O 操作。
- (5) 业务线程池从任务队列中获取消息，并发执行业务逻辑。

通过对 Netty 3.X 的 Upstream ChannelEvent 进行分析我们可以看出，Upstream 操作相关的 Handler 都是由 Netty 的 I/O work 线程负责执行的。

下面继续分析 Downstream ChannelEvent 相关线程模型，如图 9-7 所示。

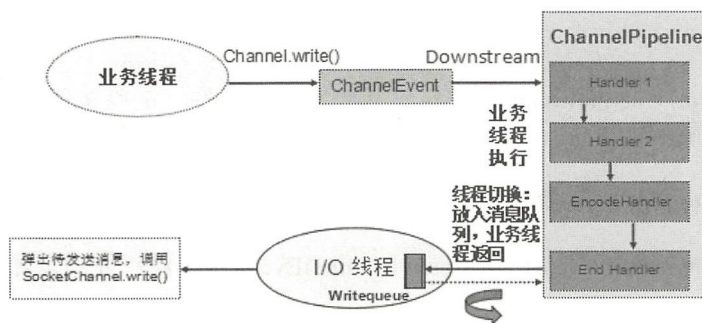


图 9-7 Netty 3.X 的 Downstream ChannelEvent 相关线程模型

从图 9-7 中可以看出，Downstream ChannelEvent 的主要处理流程如下。

- (1) 业务线程发起 Channel 的写操作，发送消息。
- (2) Netty 将写操作封装成 Downstream ChannelEvent，触发事件向下传播。
- (3) 写事件被调度到 ChannelPipeline 中，由业务线程按照 Handler 链串行调用支持 Downstream 事件的 ChannelHandler。





(4) 执行到最后一个 Handler，将编码后的消息提供给发送队列，业务线程返回。

(5) Netty 的 I/O 线程从发送消息队列中取出消息，调用 SocketChannel 的 write 方法进行消息发送。

对于 Netty 3.X，在消息发送时由业务线程执行 ChannelHandler，这就意味着 ChannelHandler 会被多线程并发调用。

## 9.5.2 Netty 4.X 版本线程模型

Netty 4.X 的线程模型相比 3.X 更简单一些，所有的 I/O 操作，无论是 Netty I/O 线程发起的，还是业务线程发起的，都统一由 Netty 的 NioEventLoop 线程执行（备注：业务没为 ChannelHandler 额外绑定线程），Netty 4.X 的 Inbound 和 Outbound 操作线程模型如图 9-8 所示。

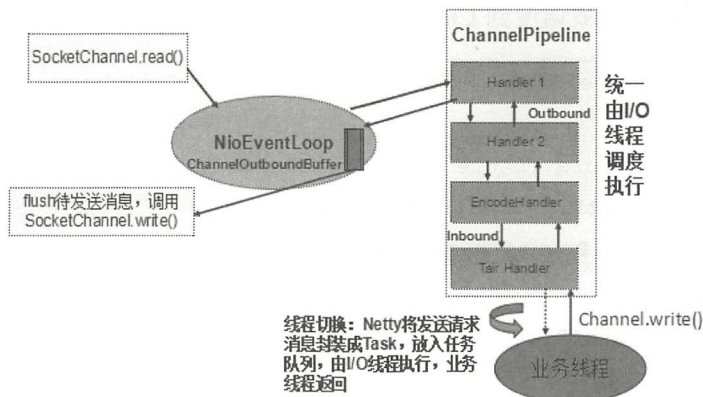


图 9-8 Netty 4.X 的 Inbound 和 Outbound 操作线程模型

从图 9-8 中可以看出，Outbound 操作的主要处理流程如下。

(1) I/O 线程 NioEventLoop 从 SocketChannel 中读取数据报，将 ByteBuffer 投递到 ChannelPipeline，触发 ChannelRead 事件。

(2) I/O 线程 NioEventLoop 调用 ChannelHandler 职责链，直到将消息投递到业务线程，然后 I/O 线程返回，继续后续的读写操作。





(3) 业务线程调用 `ChannelHandlerContext.write(Object msg)` 方法进行消息发送。

(4) 如果是由业务线程发起的写操作，`ChannelHandlerInvoker` 将发送消息封装成任务，放入 I/O 线程 `NioEventLoop` 的任务队列，由 `NioEventLoop` 在 `Selector` 轮询中统一调度和执行。放入任务队列后，业务线程返回。

(5) I/O 线程 `NioEventLoop` 调用 `ChannelHandler` 职责链，进行消息发送，处理 `Outbound` 事件，直到将消息放入发送队列，然后唤醒 `Selector` 执行写操作。

通过以上流程分析，我们发现 Netty 4.X 修改了线程模型，无论是 `Inbound` 还是 `Outbound` 操作，统一由 I/O 线程 `NioEventLoop` 来调度执行。

### 9.5.3 线程模型变化点源码分析

首先对 Netty 3.X（以 Netty 3.10.0 为例）消息发送线程模型进行分析，调用 `Channel` 的 `write` 方法，消息最终被调度到 `Channels` 的 `write` 方法中，源码如下（`Channels` 类）：

---

```
public static ChannelFuture write(Channel channel, Object message, SocketAddress
remoteAddress) {
    ChannelFuture future = future(channel);
    channel.getPipeline().sendDownstream(
        new DownstreamMessageEvent(channel, future, message,
remoteAddress));
    return future;
}
```

---

将需要发送的消息封装成 `DownstreamMessageEvent`，然后调用 `ChannelPipeline` 的 `sendDownstream` 方法，代码如下：

---

```
public void sendDownstream(ChannelEvent e) {
    DefaultChannelHandlerContext tail = getActualDownstreamContext(this.tail);
    if (tail == null) {
        try {
            getSink().eventSunk(this, e);
            return;
        }
    }
}
```

---





```
    } catch (Throwable t) {  
        notifyHandlerException(e, t);  
        return;  
    }  
}  
  
sendDownstream(tail, e);  
}
```

判断当前是否已经没有需要执行的 Downstream ChannelHandler，如果没有则执行 ChannelSink 的 eventSunk 方法，判断事件类型，如果是 MessageEvent，插入发送队列，由 Netty I/O 线程异步执行，业务调用线程返回，源码如下（NioServerSocketPipelineSink 类）：

```
private static void handleAcceptedSocket(ChannelEvent e) {  
    //代码省略  
    } else if (e instanceof MessageEvent) {  
        MessageEvent event = (MessageEvent) e;  
        NioSocketChannel channel = (NioSocketChannel) event.getChannel();  
        boolean offered = channel.writeBufferQueue.offer(event);  
        assert offered;  
        channel.worker.writeFromUserCode(channel);  
    }  
}
```

如果有可以执行的 Downstream ChannelHandler，则遍历 ChannelPipeline，由业务调用方线程（备注：Netty 的 I/O 线程也可以发起 write 操作，在实际项目中通常由业务线程发起）执行对应的 ChannelHandler，代码如下（DefaultChannelPipeline 类）：

```
void sendDownstream(DefaultChannelHandlerContext ctx, ChannelEvent e) {  
    if (e instanceof UpstreamMessageEvent) {  
        throw new IllegalArgumentException("cannot send an upstream event  
to downstream");  
    }  
    try {  
        ((ChannelDownstreamHandler) ctx.getHandler()).handleDownstream(ctx, e);  
    }
```







```
    } catch (Throwable t) {  
        e.getFuture().setFailure(t);  
        notifyHandlerException(e, t);  
    }  
}
```

---

对于 Netty 4.X，在调用 `ChannelHandlerContext` 或者 `Channel` 的 `write` 操作时，会对当前的执行线程做判断，由下一个将要执行的 `ChannelHandler` 绑定的 `EventExecutor` 线程来执行，由于业务通常不需要额外为 `ChannelHandler` 绑定 `EventExecutor`，所以默认的就是当前 `Channel` 注册的 `NioEventLoop` 线程（与 `read` 操作是同一个线程），将发送消息封装成 `WriteTask`，加入 `NioEventLoop` 线程的任务队列异步执行，当前调用方业务线程返回，代码如下（`AbstractChannelHandlerContext` 类）：

---

```
private void write(Object msg, boolean flush, ChannelPromise promise) {  
    //代码省略  
} else {  
    AbstractWriteTask task;  
    if (flush) {  
        task = WriteAndFlushTask.newInstance(next, m, promise);  
    } else {  
        task = WriteTask.newInstance(next, m, promise);  
    }  
    safeExecute(executor, task, promise, m);  
}  
}
```

---

#### 9.5.4 线程模型变化总结

在进行新老版本线程模型对比之前，首先还是要熟悉一下线程串行化设计的理念。

我们知道在系统运行过程中，如果频繁地进行线程上下文切换，会带来额外的性能损耗。多线程并发执行某个业务流程，业务开发者还需要时刻对线程安全保持警惕，比如哪些数据可能会被并发修改，以及如何保护，这不仅降低了开发效率，也会带来额外的性能损耗。





为了解决上述问题，Netty 4.X 采用了串行化设计理念，从消息的读取、编码到后续 Handler 的执行，始终都由 I/O 线程 `NioEventLoop` 负责，这就意味着整个流程不会进行线程上下文的切换，数据也不会面临被并发修改的风险，对于用户而言，甚至不需要了解 Netty 的线程细节，这确实是个非常好的设计理念，如图 9-9 所示。

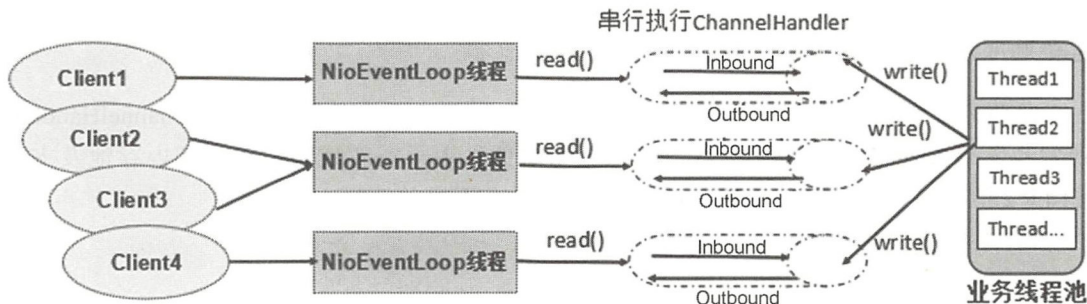


图 9-9 Netty 4.X 串行化设计理念（服务端）

一个 `NioEventLoop` 聚合了一个多路复用器 `Selector`，因此可以处理成百上千的客户端连接。Netty 的处理策略是，每当有一个新的客户端接入，则从 `NioEventLoop` 线程组中顺序获取一个可用的 `NioEventLoop`，当到达数组上限时，重新返回到 0。通过这种方式，可以基本保证各个 `NioEventLoop` 的负载均衡。一个客户端连接只注册到一个 `NioEventLoop`，这样就避免了多个 I/O 线程并发操作它。

Netty 通过串行化设计理念降低了用户的开发难度，提高了处理性能。利用线程组实现了多个串行化线程并行执行，线程之间并没有交集，这样既可以充分利用多核提升并行处理能力，又避免了线程上下文的切换和并发保护带来的额外性能损耗。

了解了 Netty 4.X 的串行化设计理念之后，我们继续看 Netty 3.X 线程模型存在的问题，总结起来，它的主要问题如下。

(1) Upstream 和 Downstream 都是 I/O 相关的操作，它们的线程模型却不统一，这给用户带来了额外的学习和使用成本。

(2) Downstream 事件由业务线程执行，通常业务会使用线程池并行处理业务消息，这就意味着在某一个时刻会有多个业务线程同时操作 `ChannelHandler`，用户需要对 `ChannelHandler` 进行并发保护（线程安全设计）。如果并发保护不当，可能会产生严重的性能瓶颈，这对开发者的技能要求非常高，降低了开发效率。





(3) 在 Downstream 事件操作过程中，例如消息编码异常，会产生 Exception，它会被转换成 Upstream 的 Exception 并通知 ChannelPipeline，这就意味着业务线程发起了 Upstream 操作。它打破了 Upstream 操作通常由 I/O 线程操作的惯例，如果开发者按照 Upstream 操作只由一个 I/O 线程执行的约束进行设计，则会产生线程并发访问安全问题。由于该场景只在特定异常发生时出现，因此错误非常隐蔽。一旦在生产环境中产生此类线程并发问题，定位难度和成本都非常高。

从整体上看，Netty 4.X 的线程模型更加归一，如果业务不使用全局共享的 ChannelHandler，通常也不需要担心线程并发问题，编码更加简单。由于在新版本中引入了内存池及支持 HTTP/2 等新特性，因此升级到 Netty 4.1.X 是一个不错的选择。

## 9.6 总结

就 Netty 而言，掌握线程模型的重要性不亚于熟练使用它的 API 和功能。很多时候业务遇到的功能、性能等问题，都是由于缺乏对 Netty 线程模型和原理的理解导致的。对 Netty 的版本升级需要从功能、兼容性和性能等多个角度进行综合考虑，切不可只盯着 API 和功能变更这个“芝麻”，而丢掉了线程模型和性能这个“西瓜”。API 的变更会导致编译错误，但是性能下降却隐藏于无形之中，稍不注意就会中招。对于强调快速交付和敏捷开发的互联网类应用，升级的时候尤其要小心，不能功能调通后简单验证就匆忙上线。





## 第 10 章

---

# Netty 并发失效导致性能下降案例

为了提升性能，如果用户实现的 `ChannelHandler` 包含复杂或者可能导致同步阻塞的业务逻辑，例如数据库操作、同步的第三方服务调用等，往往需要通过线程池来提升并发处理能力，线程池的添加有两种策略：用户自定义线程池执行业务 `ChannelHandler`，以及通过 Netty 的 `EventExecutorGroup` 机制来并行执行 `ChannelHandler`。

由于 Netty 线程池的实现比较复杂，种类也较多，如果不能深入掌握 Netty 的线程池运行原理，往往容易理解错误，进而引发各种问题。

### 10.1 业务 `ChannelHandler` 无法并发执行问题

---

业务通过 Netty 的 `DefaultEventExecutorGroup` 并行执行 `LogicServerHandler`，做性能测试时发现服务端的处理能力非常差，感觉多线程并没有生效，由于对 Netty 的线程池机制不太熟悉，所以不清楚具体原因。

#### 10.1.1 服务端并发设计相关代码分析

---

服务端没有采用传统的业务线程池执行 `LogicServerHandler`，而是使用 Netty 内置的





DefaultEventExecutorGroup 来并行调用业务的 LogicServerHandler，相关代码示例如下（简化版）：

---

```
public final class ConcurrentPerformanceServer {
    static final int PORT = Integer.parseInt(System.getProperty("port",
"18088"));

    static final EventExecutorGroup executor = new DefaultEventExecutorGroup(100);

    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    public void initChannel(SocketChannel ch) throws Exception {
                        ChannelPipeline p = ch.pipeline();
                        p.addLast(executor, new ConcurrentPerformanceServerHandler());
                    }
                });
        } catch {}
    }
}
```

//后续代码省略

---

在服务端初始化时创建了一个线程数为 100 的 EventExecutorGroup，并将其绑定到业务的 ConcurrentPerformanceServerHandler，这样就可以实现 I/O 线程和业务逻辑处理线程的隔离，同时还能并发执行 ConcurrentPerformanceServerHandler，提升性能。

在业务的 ConcurrentPerformanceServerHandler 中，通过随机的休眠来模拟复杂的业务逻辑操作耗时，同时利用定时任务线程池周期性地统计服务端的处理性能，相关代码如下：

---

```
public class ConcurrentPerformanceServerHandler extends ChannelInbound-
HandlerAdapter {
    AtomicInteger counter = new AtomicInteger(0);
    static ScheduledExecutorService scheduledExecutorService =
```

---





```

Executors.newSingleThreadScheduledExecutor();
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    scheduledExecutorService.scheduleAtFixedRate(() ->
    {
        int qps = counter.getAndSet(0);
        System.out.println("The server QPS is : " + qps);
    }, 0, 1000, TimeUnit.MILLISECONDS);
}

public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ((ByteBuf)msg).release();
    counter.incrementAndGet();
    //业务逻辑处理, 模拟业务访问 DB、缓存等, 时延在 100ms~1000ms 之间
    Random random = new Random();
    try
    {
        TimeUnit.MILLISECONDS.sleep(random.nextInt(1000));
    } catch (Exception e)
    {
        e.printStackTrace();
    }
}

//后续代码省略
}

```

---

模拟性能测试, 在客户端和服务端之间建立一个 TCP 长连接, 以 100 QPS 的速度压测服务端, 客户端代码如下 (ConcurrentPerformanceClientHandler 类):

```

public class ConcurrentPerformanceClientHandler extends ChannelInboundHandlerAdapter {
    static ScheduledExecutorService scheduledExecutorService =
Executors.newSingleThreadScheduledExecutor();
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        scheduledExecutorService.scheduleAtFixedRate(() ->
        {
            for(int i = 0; i < 100; i++)

```

```

{
    ByteBuf firstMessage = Unpooled.buffer
(ConcurrentPerformanceClient.MSG_SIZE);

    for (int k = 0; k < firstMessage.capacity(); k++) {
        firstMessage.writeByte((byte) i);
    }

    ctx.writeAndFlush(firstMessage);
}

},0,1000, TimeUnit.MILLISECONDS);
}

//后续代码省略

```

测试结果如图 10-1 所示,吞吐量是个位数,考虑到业务处理逻辑耗时在 100ms~1000ms,因此怀疑业务 ChannelHandler 并没有被并发执行,而是被单线程执行。

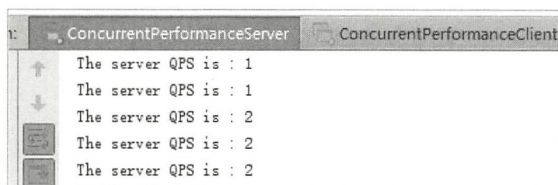


图 10-1 测试结果

为了验证猜想,查看当前服务端的线程堆栈,结果发现线程组中只有一个 DefaultEventExecutor 线程在运行,如图 10-2 所示。

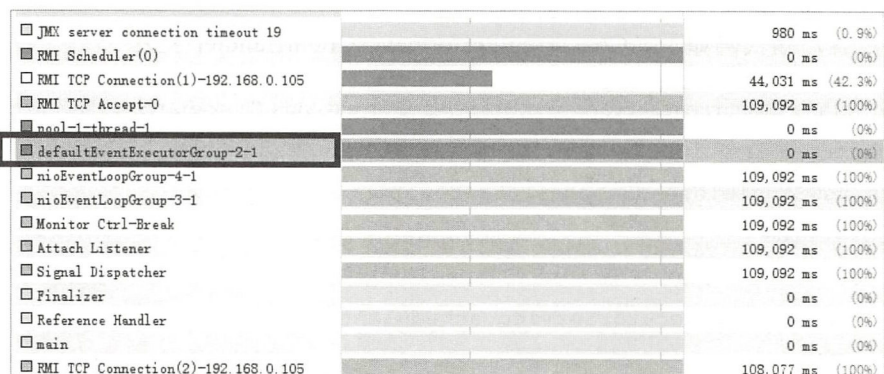


图 10-2 服务端的线程堆栈

从采集的线程运行数据看，业务设置的包含 100 个线程的 `DefaultEventExecutorGroup` 只有一个线程在运行。由于单个线程执行包含复杂业务逻辑操作的 `ChannelHandler`，因此性能不高，需要继续分析导致该问题的原因。

### 10.1.2 无法并行执行的 `EventExecutorGroup`

对源码进行分析，首先查看绑定 `DefaultEventExecutorGroup` 到业务 `ChannelHandler` 的代码，如下所示（`DefaultChannelPipeline` 类）：

---

```
public final ChannelPipeline addLast(EventExecutorGroup group, String name,
ChannelHandler handler) {
    final AbstractChannelHandlerContext newCtx;
    synchronized (this) {
        checkMultiplicity(handler);

        newCtx = newContext(group, filterName(name, handler), handler);
        addLast0(newCtx);
    }
    //后续代码省略
}
```

---

创建 `DefaultChannelHandlerContext`，调用 `childExecutor(group)` 方法，从 `EventExecutorGroup` 中选择一个 `EventExecutor` 绑定到 `DefaultChannelHandlerContext`，相关代码如下：

---

```
private EventExecutor childExecutor(EventExecutorGroup group) {
    //代码省略

    EventExecutor childExecutor = childExecutors.get(group);
    if (childExecutor == null) {
        childExecutor = group.next();
        childExecutors.put(group, childExecutor);
    }
    return childExecutor;
}
```

---

通过 `group.next()` 方法，从 `EventExecutorGroup` 中选择一个 `EventExecutor`，存放到 `EventExecutor` `Map` 中，选择 `EventExecutor` 的具体实现是调用 `GenericEventExecutorChooser` 的 `next()` 方法，

代码如下（GenericEventExecutorChooser 类）：

---

```
public EventExecutor next() {
    return executors[Math.abs(idx.getAndIncrement()) % executors.length];
}
```

---

通过分析以上代码，我们发现对于某个具体的 TCP 连接，绑定到业务 ChannelHandler 实例上的线程池为 DefaultEventExecutor，下面分析当接收到请求消息时，业务 ChannelHandler 的调用情况，代码如下（AbstractChannelHandlerContext 类）：

---

```
static void invokeChannelRead(final AbstractChannelHandlerContext next,
    Object msg) {
    final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg,
        "msg"), next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        next.invokeChannelRead(m);
    } else {
        executor.execute(new Runnable() {
            @Override
            public void run() {
                next.invokeChannelRead(m);
            }
        });
    }
}
```

---

业务 ChannelHandler 绑定的 EventExecutor 为 DefaultEventExecutor，因此调用的就是 DefaultEventExecutor 的 execute 方法，由于 DefaultEventExecutor 继承自 SingleThreadEventExecutor，所以执行 execute 方法就是把 Runnable 放入任务队列由单线程执行，DefaultEventExecutor 的 execute 方法如图 10-3 所示。

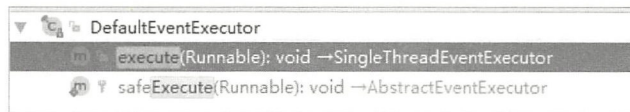


图 10-3 DefaultEventExecutor 的 execute 方法

通过以上分析得知，对于某个 Channel 对应的业务 ChannelHandler 实例，无论消费端有多少个线程来并发压测某条链路，对于服务端都只有一个 DefaultEventExecutor 线程来运行业务 ChannelHandler，无法实现并行调用，业务 ChannelHandler 的线程调度模型如图 10-4 所示。

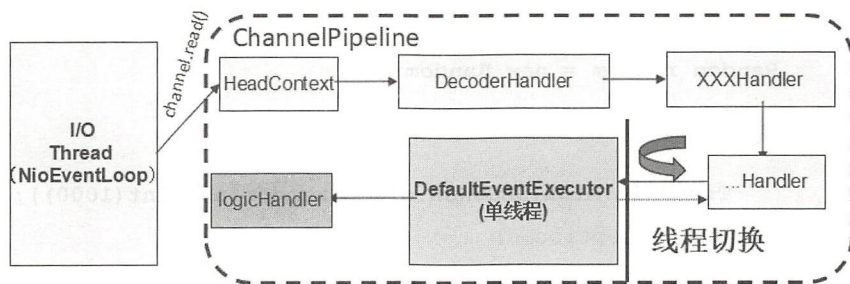


图 10-4 业务 ChannelHandler 的线程调度模型

### 10.1.3 并行执行优化策略和结果

针对上述问题代码，按照业务场景，有两种优化策略，具体如下。

(1) 按照真实的业务组网，如果所有客户端的并发连接数（TCP/HTTP 链路数）小于业务需要配置的线程数，则建议将请求消息封装成任务，投递到后端业务线程池执行，ChannelHandler 不需要处理复杂业务逻辑，也不需要再绑定 EventExecutorGroup。

(2) 如果所有客户端的并发连接数（TCP/HTTP 链路数）大于或者等于业务需要配置的线程数，则可以为业务 ChannelHandler 绑定 EventExecutorGroup，并在业务 ChannelHandler 中执行各种业务逻辑（包含 DB 访问等可能导致阻塞或者增加耗时的复杂业务逻辑操作）。

优化策略（1）的示例代码如下，客户端仍然是通过单 TCP 链路、以 100 QPS 的速度压测服务端，在服务端后台启动业务线程池，并发执行复杂的业务逻辑操作（ConcurrentPerformanceServerHandlerV2 类）。

```

public class ConcurrentPerformanceServerHandlerV2 extends
ChannelInboundHandlerAdapter {
    static ExecutorService executorService = Executors.newFixedThreadPool(100);

```



```

public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ((ByteBuf)msg).release();
    executorService.execute(() ->
    {
        counter.incrementAndGet();
        //业务逻辑处理，模拟业务访问 DB、缓存等，时延在 100ms~1000ms 之间
        Random random = new Random();
        try
        {
            TimeUnit.MILLISECONDS.sleep(random.nextInt(1000));
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    });
}

```

优化策略（1）运行结果如图 10-5 所示，服务端的处理性能基本维持在 100 QPS，与客户端的发送速度相等，说明尽管服务端有复杂的业务逻辑操作，但是由于并发 100 个线程执行，服务端的处理性能仍然能够满足客户端的压测速度需要。

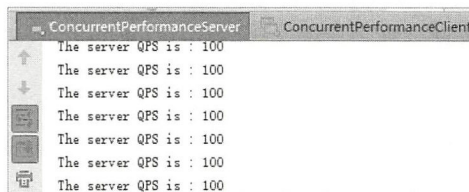


图 10-5 优化策略（1）运行结果

优化策略（2）为：客户端创建 100 个 TCP 连接，每个连接每秒发送 1 条请求消息，则整体 QPS 也为 100，此时服务端采用业务 ChannelHandler 绑定 EventExecutorGroup 的方案进行测试，相关代码如下（MulChannelPerformanceClient 类）。

```

public void connect() throws Exception
{
    EventLoopGroup group = new NioEventLoopGroup(8);

```

```

Bootstrap b = new Bootstrap();
b.group(group)
  .channel(NioSocketChannel.class)
  .option(ChannelOption.TCP_NODELAY, true)
  .handler(new ChannelInitializer<SocketChannel>() {
      @Override
      public void initChannel(SocketChannel ch) throws Exception {
          ch.pipeline().addLast(new ConcurrentPerformanceClient-
HandlerV2());
      }
  });

ChannelFuture f = null;
for(int i = 0; i < 100; i++)
    f = b.connect(HOST, PORT).sync();
//后续代码省略
}

```

优化策略（2）运行结果如图 10-6 所示，服务端的 QPS 维持在 100 左右，说明业务 ChannelHandler 被并行调用，性能达到了预期目标。

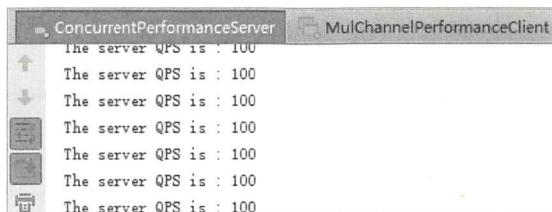


图 10-6 优化策略（2）运行结果

## 10.2 Netty DefaultEventExecutor 工作机制

DefaultEventExecutor 是 SingleThreadEventExecutor 的一个实现子类，与其对应的还有 NioEventLoop 类。DefaultEventExecutor 实际是一个典型的 Event Reactive Thread 实现，各种任务被加入任务队列，由一个工作线程循环执行。它与 NioEventLoop 类的主要差别是，

一个侧重于处理网络 I/O 相关的各种事件，例如连接操作、消息读取和发送操作，另一个侧重于处理业务相关的逻辑操作，例如将业务的 ChannelHandler 与具体的 DefaultEventExecutor 绑定起来，由 DefaultEventExecutor 异步执行业务逻辑，实现网络 I/O 线程与业务逻辑处理线程的分离。

## 10.2.1 DefaultEventExecutor 原理和源码分析

DefaultEventExecutor 是 JDK 线程池 ExecutorService 的一种优化实现（针对通信框架领域），它的类继承关系如图 10-7 所示。

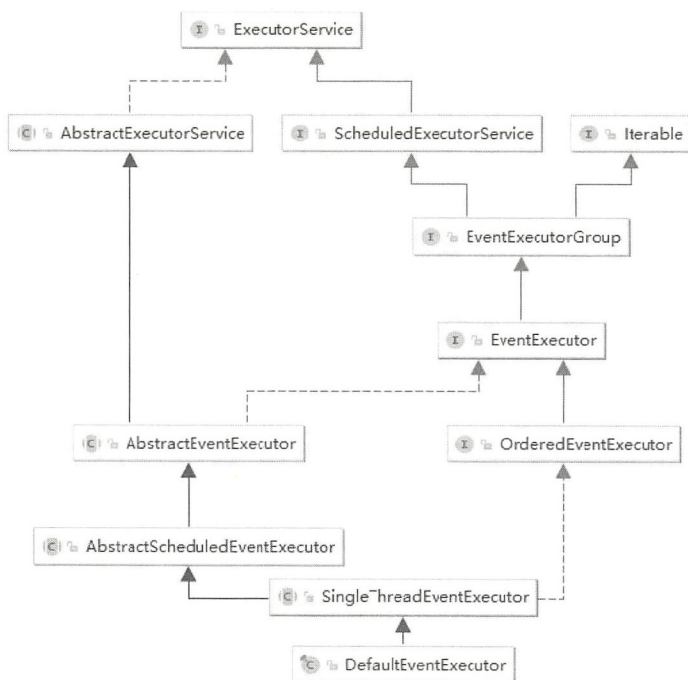


图 10-7 DefaultEventExecutor 的类继承关系

由于同时实现了 ScheduledExecutorService 和 ExecutorService 接口，所以与 NioEventLoop 线程类似，DefaultEventExecutor 可以同时处理普通的 Runnable 和定时任务，相比 NioEventLoop，它的 run 方法实现更加简单，从任务队列中循环获取 Runnable 并执行，代码如下（DefaultEventExecutor 类）：

---

```
protected void run() {
    for (;;) {
        Runnable task = takeTask();
        if (task != null) {
            task.run();
            updateLastExecutionTime();
        }
        if (confirmShutdown()) {
            break;
        }
    }
}
```

---

对于 `NioEventLoop`，除了处理任务队列、定时任务，还需要同时轮询 `Selector`，处理网络 I/O 相关操作，并对双方的执行时间进行控制，防止一方执行时间过长影响另一方的运行。

### 10.2.2 业务线程池优化策略

---

当 Netty 服务端接入的客户端连接数比较多时，使用 `EventExecutorGroup` 绑定业务 `ChannelHandler` 或者在后台创建一个 JDK 线程池专门处理业务逻辑操作都是可行的，究竟哪种方式更优呢？

下面对两种不同的并行调用线程模型进行分析，首先看业务绑定 `EventExecutorGroup` 模式的线程模型（以 I/O 线程数等于业务线程数为例，在实际业务中业务线程数往往大于 I/O 线程数，所以一个 I/O 线程可能对应一个或者多个业务线程），如图 10-8 所示。

对于某个客户端连接 `Channel`，只会注册到一个 `NioEventLoop` 线程中，用于处理网络 I/O 操作，业务的 `ChannelHandler` 指定了运行线程池 `EventExecutorGroup` 之后，创建 `ChannelHandlerContext` 上下文时会从 `EventExecutorGroup` 中选择一个 `EventExecutor` 绑定到该 `Channel` 对应的 `ChannelHandler` 实例。对于某个 `Channel`，它的两侧都进行了线程绑定，消息接收线程是 `NioEventLoop`，业务逻辑处理在 `EventExecutor` 线程中，这样就实现了网络 I/O 线程与业务逻辑处理线程的绑定，对于某个 TCP 连接由于双方是一一对一的关系，

所以降低了锁竞争。当客户端并发连接比较多时，会有  $N$  个 Channel 被并行处理，这样既可以充分利用多核 CPU 的计算资源，又最大程度地降低了锁竞争，提升了系统的并发处理能力。

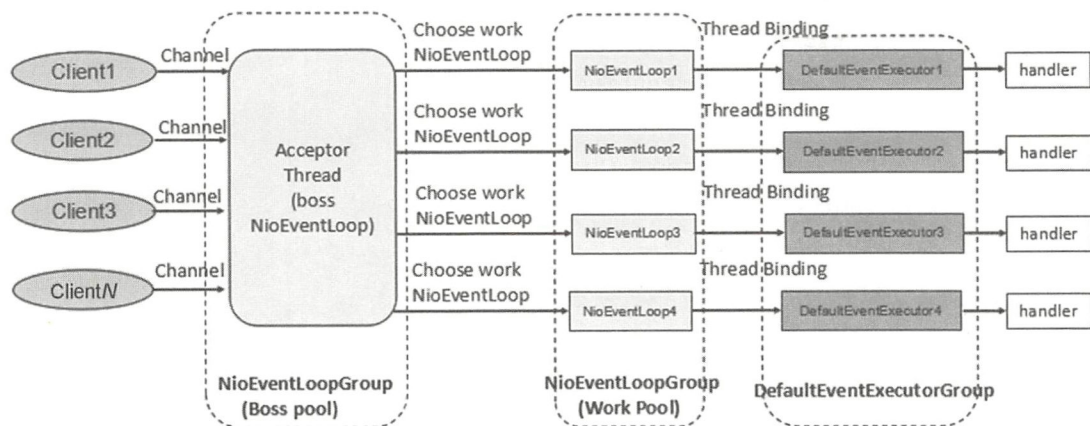


图 10-8 业务绑定 EventExecutor 模式的线程模型

采用后端创建一个统一的 JDK 线程池做业务逻辑处理方案时，系统线程模型如图 10-9 所示。

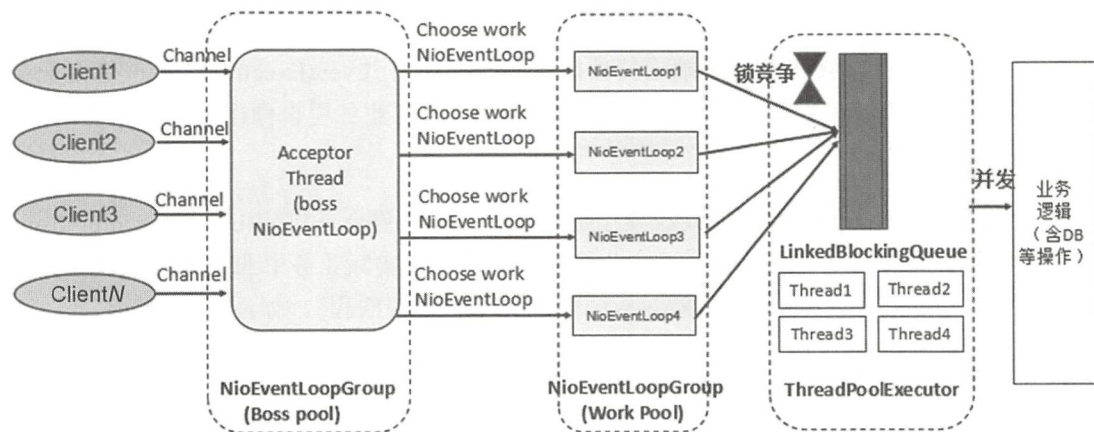


图 10-9 业务自定义线程池模式的线程模型

从图 10-9 可以看出，当后端创建一个统一的业务逻辑处理线程池时，网络 I/O 处理线程 NioEventLoop（多个）与业务线程池就会形成  $M$  对  $N$  的映射关系，由于 JDK 的



ThreadPoolExecutor 采用的是“一个阻塞队列+N个工作线程”的模型，如果业务线程数比较多，就会形成激烈的锁竞争，相关代码如下（ThreadPoolExecutor 类）：

---

```
protected void run() {
    public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        if (isRunning(c) && workQueue.offer(command)) {
            //后续代码省略
        }
    }
}
```

---

尽管 LinkedBlockingQueue 通过读写锁来提升性能，但是当业务线程数和写操作比较多时，锁竞争对性能的影响还是非常大的。

由以上分析得知，当客户端并发接入数比较多时，可以利用 Netty 提供的网络 I/O 线程和 ChannelHandler 执行线程绑定机制来降低锁竞争，提升系统性能。当然，如果业务自定义线程池自己实现并且做了锁竞争优化，也可以达到同样的优化效果。清楚原理之后，优化措施也是多样化的，不必拘泥于某一种实现。

如果业务采用自定义线程池，优化方向是尽量消除锁竞争，其中一种优化思路如图 10-10 所示。

关键技术点如下（以 RPC 框架私有协议长连接为例）。

（1）利用 Netty 的 ChannelId 绑定业务线程池的某个业务线程，后续该 Channel 的所有消息读取和发送都由绑定的 Netty NioEventLoop 和业务线程来执行，把锁竞争降到最低。

（2）业务线程池采用一个线程对应一个消息队列的方式，降低队列的锁竞争。可以继承 JDK 的 ExecutorService 自己实现，或者利用 Executors 的 newSingleThreadExecutor()方法创建多个 SingleThreadExecutor，这样就实现了工作线程和消息队列的一对一关系。

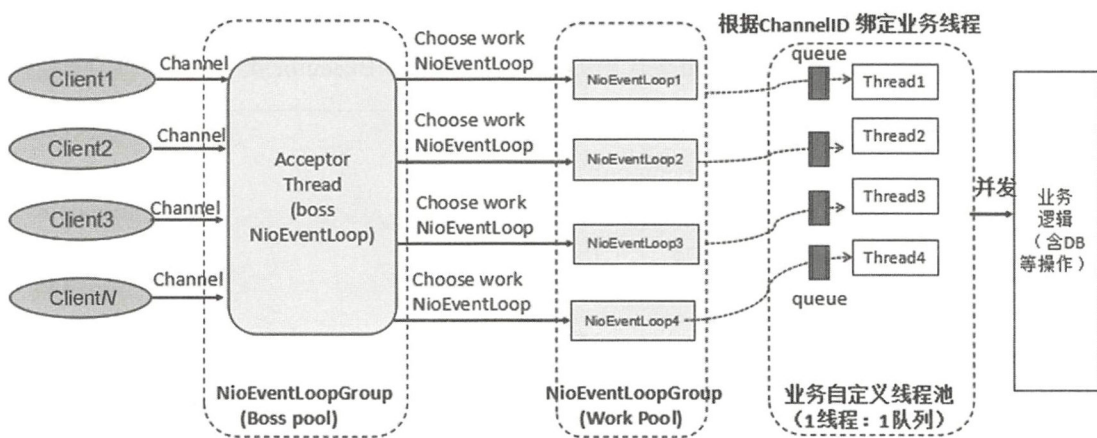


图 10-10 业务自定义线程池的一种优化思路

### 10.2.3 Netty 线程绑定机制原理和源码分析

首先看 NioEventLoop 线程与 SocketChannel 绑定关系的创建，代码如下 (ServerBootstrapAcceptor 类)：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    //代码省略
    try {
        childGroup.register(child).addListener(new
ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future)
throws Exception {
                if (!future.isSuccess()) {
                    forceClose(child, future.cause());
                }
            }
        });
    } catch (Throwable t) {
        forceClose(child, t);
    }
}
```

```

    }
}
}

```

当 Netty 服务端接收客户端 TCP 连接时, 触发 `channelRead` 方法, 由 `ServerBootstrapAcceptor` 负责将新接入的 `SocketChannel` 注册到 `NioEventLoop` 的 `Selector`, 同时建立 `SocketChannel` 和 `NioEventLoop` 的绑定关系, 代码如下 (`MultithreadEventLoopGroup` 类):

```

public ChannelFuture register(Channel channel) {
    return next().register(channel);
}

```

`next` 方法负责从 `NioEventLoopGroup` 中选择一个 `NioEventLoop` 线程做 `SocketChannel` 的注册, 选择算法请参考 `GenericEventExecutorChooser` 类。

业务 `ChannelHandler` 异步执行的线程切换点发生在 `AbstractChannelHandlerContext` 的 `invokeChannelRead(final AbstractChannelHandlerContext next, Object msg)` 方法中, 由 Netty 的 `NioEventLoop` 线程切换到业务 `ChannelHandler` 绑定的 `DefaultEventExecutor` 中。

切换之前, 由 Netty 的 I/O 线程 `NioEventLoop` 执行 `ChannelHandler`, 如图 10-11 所示。

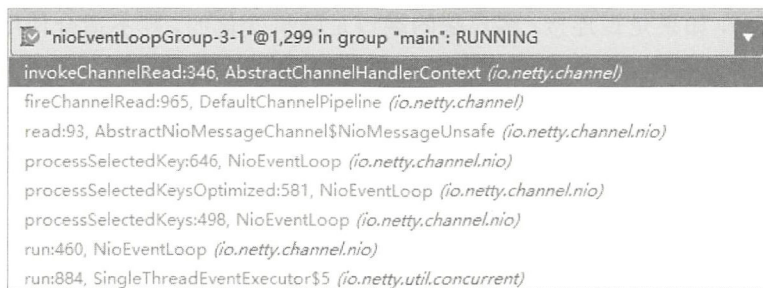


图 10-11 切换之前 I/O 线程运行堆栈

由于业务 `ChannelHandler` 绑定了 `DefaultEventExecutor`, 因此业务 `ChannelHandler` 的执行被切换到与 `Channel` 绑定的 `DefaultEventExecutor` 线程中, 如图 10-12 所示。

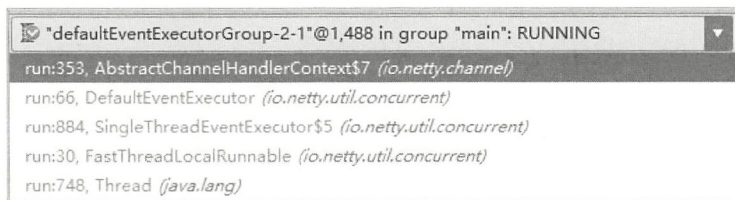


图 10-12 由 I/O 线程切换到绑定的 DefaultEventExecutor 线程

## 10.3 总结

Netty 框架本身实现了高性能的网络读写操作，但是后端业务逻辑执行却是影响性能的关键要素，如果直接将复杂的业务逻辑操作放在 I/O 线程中完成，一些同步阻塞操作可能会导致 I/O 线程被阻塞。当把业务逻辑单独拆分到业务线程池中进行处理，与 I/O 线程隔离时，不同的业务线程模型对性能的影响也非常大。Netty 提供了默认的并行调度 ChannelHandler 的能力，但是如果使用不当，也会带来性能问题。对于业务自定义实现的线程池，如果追求更高的性能，就需要在消除或者减轻锁竞争上下工夫，线程绑定技术是一个不错的选择，但是也需要根据业务实际场景来实现，例如 TCP 长连接就可以使用 ChannelId 做 Key，如果是短连接，客户端的端口是随机变化的，则不适合使用 ChannelId。

## 第 11 章

---

# IoT 百万长连接性能调优案例

随着车联网、智能家居、智慧城市等业务的发展，IoT 进入了飞速发展期。由于要接入海量的硬件设备和传感器，且协议多样化，同时还要在极短的时间内处理大量的数据，所以对服务端的协议接入和处理能力要求极高。

由于 Netty 内置了多种协议栈（TCP、HTTP、MQTT 等），同时利用它提供的编解码框架可以快速地完成私有协议的接入，因此 Netty 在 IoT 领域得到了较广泛的应用。基于 Netty 构建的物联网服务端，实现更多设备的接入和数据收发，提升性能，是一个巨大的挑战。

### 11.1 海量长连接接入面临的挑战

---

当客户端的并发连接数达到数十万或者数百万时，系统一个较小的抖动就会导致很严重的后果，例如服务端的 GC，导致应用暂停（STW）的 GC 持续几秒，就会导致海量的客户端设备掉线或者消息积压，一旦系统恢复，会有海量的设备接入或者海量的数据发送，很可能瞬间就把服务端冲垮。



### 11.1.1 IoT 设备接入特点

---

IoT 设备接入有如下几个特点（以车联网为例）。

- （1）使用的网络主要是运营商的无线移动网络，网络质量不稳定，例如在一些偏远地区、丘陵地带等信号很差，网络容易闪断。
- （2）海量的端侧设备接入，而且通常使用长连接，服务端的压力很大。
- （3）不稳定，消息丢失、重复发送、延迟送达、过期发送时有发生。
- （4）协议不统一，有各种私有协议，开发和测试成本较高。

### 11.1.2 IoT 服务端性能优化场景

---

服务端的性能优化主要有如下几类场景。

- （1）降成本：对于很多初创型公司，资金和资源相对比较紧张，希望服务端的单节点性能足够高，以降低硬件和运维成本。
- （2）云服务提供商：对于提供 IoT 解决方案的云服务提供商，由于部署规模比较大（假如有上千个服务节点），单节点的性能提升效益会被放大，带来的经济收益很高。
- （3）技术竞争力：在很多场景下，单个服务节点的处理性能仍然是最重要的技术指标之一，高性能对提升产品竞争力非常重要。

### 11.1.3 服务端面临的性能挑战

---

尽管都是采用 Netty 构建的，但是不同的 IoT 解决方案性能却差异很大，主要原因如下。

- （1）对协议接入相关的操作系统参数没有进行针对性的调优。
- （2）对 Netty 海量 TCP 接入的性能参数不熟悉，没有做优化。
- （3）对 JVM 的垃圾收集器没有做调优，不合理的内存设置或者 GC 参数导致 GC 频率和应用中断频发，影响系统的稳定性。

要想实现海量设备的接入，需要对操作系统相关参数、Netty 框架、JVM GC 参数，甚至业务代码做针对性的优化，各种优化要素互相影响，设置或者组合不当就容易导致性能问题，这也是服务端实现海量设备接入的最大挑战。

## 11.2 智能家居内存泄漏问题

智能家居 MQTT 消息服务中间件，保持 10 万用户在线长连接，2 万用户并发消息请求。在程序运行一段时间后，发现内存泄漏，怀疑是 Netty 的 Bug。相关信息如下。

(1) 硬件资源：MQTT 消息服务中间件服务器内存 16GB，CPU 8 核。

(2) Netty 中 boss 线程池大小为 1，worker 线程池大小为 8，其余线程分配给业务使用。该分配方式后来调整为 worker 线程池大小为 16，问题依旧。

### 11.2.1 服务端内存泄漏原因定位

首先需要 Dump 内存堆栈，对疑似内存泄漏的对象和引用关系进行分析，如图 11-1 所示。

Name	Instance count	Difference
char[ ]	3,400,328	-1,448,882 (-30 %)
java.util.HashMap\$Entry	3,956,168	+2,597,922 (+191 %)
io.netty.channel.ChannelOutboundBuffer\$Entry	2,624,096	+2,240,000 (+583 %)
java.nio.DirectByteBuffer	1,529,161	-672,671 (-31 %)
java.lang.Object[ ]	1,502,173	-343,175 (-19 %)
int[ ]	242,299	+5,121 (+2 %)
io.netty.util.concurrent.ScheduledFutureTask	1,101,894	+1,089,886 (+9076 %)
byte[ ]	644,180	-328,374 (-34 %)
io.netty.channel.DefaultChannelHandlerContext	656,009	+560,000 (+583 %)
sun.misc.Cleaner	1,135,417	-463,772 (-29 %)
java.util.HashMap\$Entry[ ]	511,599	+308,732 (+152 %)
io.netty.buffer.UnpooledUnsafeDirectByteBuf	550,545	-290,964 (-35 %)
java.lang.String	1,605,781	-607,173 (-27 %)
java.nio.DirectByteBuffer\$Deallocator	1,135,415	-463,774 (-29 %)
io.netty.util.concurrent.PromiseTask\$RunnableAdapter	1,101,886	+1,089,886 (+9082 %)

图 11-1 Dump 内存堆栈

我们发现 Netty 的 `ScheduledFutureTask` 增加了 9076%，达到 110 万个实例，通过对业务代码的分析发现，用户使用 `IdleStateHandler` 在链路空闲时进行业务逻辑处理，但是空闲时间设置得比较长，为 15 分钟。Netty 的 `IdleStateHandler` 会根据用户的使用场景，启动三类定时任务，分别是 `ReaderIdleTimeoutTask`、`WriterIdleTimeoutTask` 和 `AllIdleTimeoutTask`，它们都会被加入 `NioEventLoop` 的任务队列调度和执行。

由于超时时间过长，10 万个长连接会创建 10 万个 `ScheduledFutureTask` 对象，每个对象还保存了业务的成员变量，非常消耗内存。用户的老年代设置得比较大，一些定时任务被晋升到老年代，没有被新生代 GC 回收，导致内存一直增长，用户误认为存在内存泄漏。

事实上，通过进一步分析发现，用户的超时时间设置得非常不合理，15 分钟达不到快速检测设备是否掉线的设计目标，将超时时间设置为 45 秒后，内存可以正常回收，问题解决。

### 11.2.2 问题背后的一些思考

---

如果是 100 个长连接，即便是长周期的定时任务，也不存在内存泄漏问题，在新生代中通过 `minor GC` 就可以实现内存回收。正是因为 10 万数量级的长连接，导致小问题被放大，引发了后续的各种问题。

如果用户确实有长周期的定时任务，该如何处理？对于海量长连接的接入场景，代码处理稍有不慎，就满盘皆输，下一节我们针对 Netty 的架构特点，讲解如何对海量设备接入做性能调优。

## 11.3 操作系统参数调优

---

要实现百万级的长连接接入，首先需要对服务端的操作系统参数进行性能调优，如果保持出厂的默认配置，性能是无法满足业务需求的。

### 11.3.1 文件描述符

首先查看系统最大文件句柄数，执行命令 `# cat /proc/sys/fs/file-max`，查看最大句柄数是否满足需要，如果不满足，通过 `# vim /etc/sysctl.conf` 命令插入如下配置：

---

```
fs.file-max = 1000000
```

---

配置完成后，执行 `#sysctl -p` 命令，让配置修改立即生效。

设置完系统最大文件句柄数，对单进程打开的最大句柄数进行设置。通过 `ulimit -a` 命令查看当前设置的值是否满足要求：

---

```
[root@lilinfeng ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 256324
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files            (-n) 1024
//后续输出省略
```

---

当并发接入的 TCP 连接数超过上限时，就会提示 “too many open files”，所有新的客户端接入将失败。通过 `#vi /etc/security/limits.conf` 命令添加如下配置参数：

---

```
* soft  nofile  1000000
* hard  nofile  1000000
```

---

修改之后保存，注销当前用户，重新登录，通过 `ulimit -a` 命令查看修改是否生效。

### 11.3.2 TCP/IP 相关参数

需要重点调优的 TCP/IP 参数如下。

(1) `net.ipv4.tcp_rmem`: 为每个 TCP 连接分配的读缓冲区内存大小。第一个值是 socket 接收缓冲区分配的最小字节数。第二个值是默认值，缓冲区在系统负载不高的情况下可以增长到该值。第三个值是接收缓冲区分配的最大字节数。

(2) `net.ipv4.tcp_wmem`: 为每个 TCP 连接分配的写缓冲区内存大小。第一个值是 socket 发送缓冲区分配的最小字节数。第二个值是默认值，缓冲区在系统负载不高的情况下可以增长到该值。第三个值是发送缓冲区分配的最大字节数。

(3) `net.ipv4.tcp_mem`: 内核分配给 TCP 连接的内存，单位是 page（1 个 page 通常为 4096 字节，可以通过 `#getconf PAGESIZE` 命令查看），包括最小、默认和最大三个配置项。

(4) `net.ipv4.tcp_keepalive_time`: 最近一次数据包发送与第一次 keep alive 探测消息发送的时间间隔，用于确认 TCP 连接是否有效。

(5) `tcp_keepalive_intvl`: 在未获得探测消息响应时，发送探测消息的时间间隔。

(6) `tcp_keepalive_probes`: 判断 TCP 连接失效连续发送的探测消息个数，达到之后判定连接失效。

(7) `net.ipv4.tcp_tw_reuse`: 是否允许将 TIME\_WAIT Socket 重新用于新的 TCP 连接，默认为 0，表示关闭。

(8) `net.ipv4.tcp_tw_recycle`: 是否开启 TCP 连接中 TIME\_WAIT Socket 的快速回收功能，默认为 0，表示关闭。

(9) `net.ipv4.tcp_fin_timeout`: 套接字自身关闭时保持在 FIN\_WAIT\_2 状态的时间，默认为 60。

通过 `#vi /etc/sysctl.conf` 命令对上述网络参数进行优化，具体修改如下（大约可以接入 50 万个连接，可以根据业务需要调整参数）：

---

```
net.ipv4.tcp_mem = 64608 1048576 2097152
net.ipv4.tcp_wmem = 4096 87380 4194304
net.ipv4.tcp_rmem = 4096 87380 4194304
net.ipv4.tcp_keepalive_time = 1800
net.ipv4.tcp_keepalive_intvl = 20
net.ipv4.tcp_keepalive_probes = 5
```



```
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_fin_timeout = 30
```

---

修改完成后，通过执行 `#sysctl -p` 命令使配置立即生效。

### 11.3.3 多网卡队列和软中断

随着网络带宽的不断提升，单核 CPU 不能完全满足网卡的需求，通过多队列网卡驱动的支持，将各个队列通过中断绑定到不同的 CPU 内核，以满足对网络吞吐量要求比较高的业务场景的需要。

多队列网卡需要网卡硬件支持，首先判断当前系统是否支持多队列网卡，通过命令“`lspci -vvv`”或者“`ethtool -l 网卡 interface 名`”查看网卡驱动型号，根据网卡驱动官方说明确认当前系统是否支持多队列网卡（是否支持多队列网卡与网卡硬件、操作系统版本等有关）。有些网卡驱动默认开启了多队列网卡，有些则没有，由于不同的网卡驱动、云服务商提供的开启命令不同，因此需要根据实际情况处理，此处不再详细列举开启方式。

对于不支持多队列网卡的系统，如果内核版本支持 RPS（kernel 2.6.35 及以上版本），开启 RPS 后可以实现软中断，提升网络的吞吐量。RPS 根据数据包的源地址、目的地址及目的和源端口，算出一个 hash 值，然后根据这个 hash 值选择软中断运行的 CPU，从上层来看，也就是说将每个连接和 CPU 绑定，并通过这个 hash 值，在多个 CPU 上均衡软中断，提升网络并行处理性能，它实际提供了一种通过软件模拟多队列网卡的功能。

## 11.4 Netty 性能调优

### 11.4.1 设置合理的线程数

对于线程池的调优，主要集中在用于接收海量设备 TCP 连接、TLS 握手的 Acceptor 线程池（Netty 通常叫 boss NioEventLoopGroup）上，以及用于处理网络数据读写、心跳发送的 I/O 工作线程池（Netty 通常叫 work NioEventLoopGroup）上。

对于 Netty 服务端，通常只需要启动一个监听端口用于端侧设备接入即可，但是如果服务端集群实例比较少，甚至是单机（或者双机冷备）部署，在端侧设备在短时间内大量接入时，需要对服务端的监听方式和线程模型做优化，以满足短时间内（例如 30s）百万级的端侧设备接入的需要。

服务端可以监听多个端口，利用主从 Reactor 线程模型做接入优化，前端通过 SLB 做 4 层/7 层负载均衡。主从 Reactor 线程模型特点如下：服务端用于接收客户端连接的不再是一个单独的 NIO 线程，而是一个独立的 NIO 线程池；Acceptor 接收到客户端 TCP 连接请求并处理后（可能包含接入认证等），将新创建的 SocketChannel 注册到 I/O 线程池（sub Reactor 线程池）的某个 I/O 线程，由它负责 SocketChannel 的读写和编解码工作；Acceptor 线程池仅用于客户端的登录、握手和安全认证等，一旦链路建立成功，就将链路注册到后端 sub Reactor 线程池的 I/O 线程，由 I/O 线程负责后续的 I/O 操作。IoT 服务端主从线程池模型如图 11-2 所示。

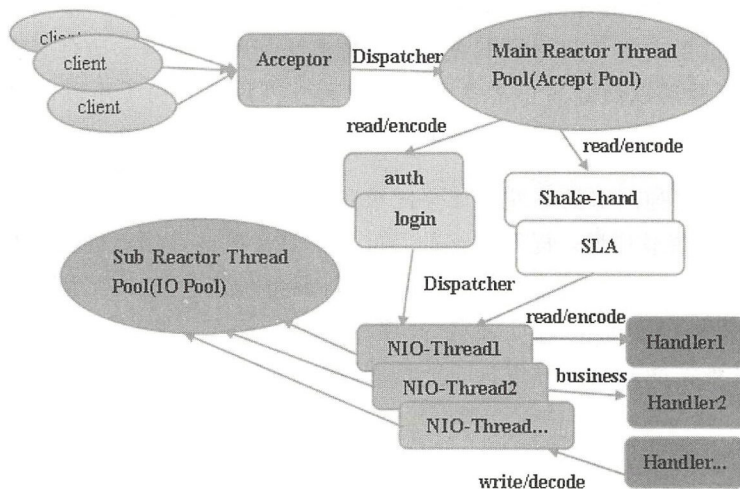


图 11-2 IoT 服务端主从线程池模型

由于同时监听了多个端口，每个 ServerSocketChannel 都对应一个独立的 Acceptor 线程，这样就能并行处理，加速端侧设备的接入速度，减小端侧设备的连接超时失败率，提升服务端单节点的处理性能。

对于 I/O 工作线程池的优化，可以先采用系统默认值（即 CPU 内核数×2）进行性能测试，在性能测试过程中采集 I/O 线程的 CPU 占用大小，看是否存在瓶颈，具体策略如下。

(1) 通过执行 `ps -ef|grep java` 找到服务端进程 pid。

(2) 执行 `top -Hp pid` 查询该进程下所有线程的运行情况，通过“shift + p”对 CPU 占用大小做排序，获取线程的 pid 及对应的 CPU 占用大小。

(3) 使用 `printf '%x\n' pid` 将 pid 转换成 16 进制格式。

(4) 通过 `jstack -f pid` 命令获取线程堆栈，或者通过 `jvisualvm` 工具打印线程堆栈，找到 I/O work 工作线程，查看它们的 CPU 占用大小及线程堆栈，如图 11-3 所示。

```

nioEventLoopGroup-4-6" #19 prio=10 cs_prio=2 tid=0x0000000016736800 nid=0x1e38 runnable [0x000000001856e000]
  java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.WindowsSelectorImpl$SubSelector.poll0(Native Method)
    at sun.nio.ch.WindowsSelectorImpl$SubSelector.poll(WindowsSelectorImpl.java:296)
    at sun.nio.ch.WindowsSelectorImpl$SubSelector.access$400(WindowsSelectorImpl.java:278)
    at sun.nio.ch.WindowsSelectorImpl.doSelect(WindowsSelectorImpl.java:159)
    a sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    - locked <0x00000000ec900f18> (a io.netty.channel.nio.SelectedSelectionKeySet)
    - locked <0x00000000ec900f30> (a java.util.Collections$UnmodifiableSet)
    - locked <0x00000000ec900e98> (a sun.nio.ch.WindowsSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at io.netty.channel.nio.SelectedSelectionKeySetSelector.select(SelectedSelectionKeySetSelector.java:62)
    at io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:755)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:410)
    at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:884)
    at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
    at java.lang.Thread.run(Thread.java:748)

```

图 11-3 I/O work 工作线程堆栈

如果连续采集几次进行对比，发现线程堆栈都停留在 `SelectorImpl.lockAndDoSelect` 处，则说明 I/O 线程比较空闲，无须对工作线程数做调整。

如果发现 I/O 线程的热点停留在读或者写操作，或者停留在 `ChannelHandler` 的执行处，则可以通过适当调大 `NioEventLoop` 线程的个数来提升网络的读写性能。调整方式有两种。

(1) 接口 API 指定：在创建 `NioEventLoopGroup` 实例时指定线程数。

(2) 系统参数指定：通过 `-Dio.netty.eventLoopThreads` 来指定 `NioEventLoopGroup` 线程池（本质是聚合了多个单线程线程池的线程组）的线程数，这种方法有个弊端，它是系统配置，即一旦设置了该参数，所有创建 `NioEventLoopGroup` 未指定线程数的地方都使用该配置，而不是默认的“CPU 内核数×2”，使用时需要注意。



## 11.4.2 心跳优化

针对海量设备接入的 IoT 服务端，心跳优化策略如下。

(1) 要能够及时检测失效的连接，并将其剔除，防止无效的连接句柄积压，导致 OOM 等问题。

(2) 设置合理的心跳周期，防止心跳定时任务积压，造成频繁的老年代 GC（新生代和老年代都有导致 STW 的 GC，不过耗时差异较大），导致应用暂停。

(3) 使用 Netty 提供的链路空闲检测机制，不要自己创建定时任务线程池，加重系统的负担，以及增加潜在的并发安全问题。

当设备突然掉电、连接被防火墙挡住、长时间 GC 或者通信线程发生非预期异常时，会导致链路不可用且不易被及时发现。特别是如果异常发生在凌晨业务低谷期间，当早晨业务高峰期到来时，由于链路不可用会导致瞬间大批量业务失败或者超时，这将对系统的可靠性产生重大的威胁。

从技术层面看，要解决链路的可靠性问题，必须周期性地对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。心跳检测机制分为三个层面。

(1) TCP 层的心跳检测，即 TCP 的 Keep-Alive 机制，它的作用域是整个 TCP 协议栈。

(2) 协议层的心跳检测，主要存在于长连接协议中，例如 MQTT。

(3) 应用层的心跳检测，它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。

心跳检测的目的就是确认当前链路是否可用，对方是否活着并且能够正常接收和发送消息。作为高可靠的 NIO 框架，Netty 也提供了心跳检测机制，心跳检测机制的工作原理如图 11-4 所示。

不同协议的心跳检测机制存在差异，归纳起来主要分为两类。

(1) Ping-Pong 型心跳：由通信一方定时发送 Ping 消息，对方接到 Ping 消息立即返回 Pong 应答消息给对方，属于“请求-响应型”心跳。

(2) Ping-Ping 型心跳：不区分心跳请求和应答，由通信双方按照约定定时向对方发送心跳 Ping 消息，它属于双向心跳。



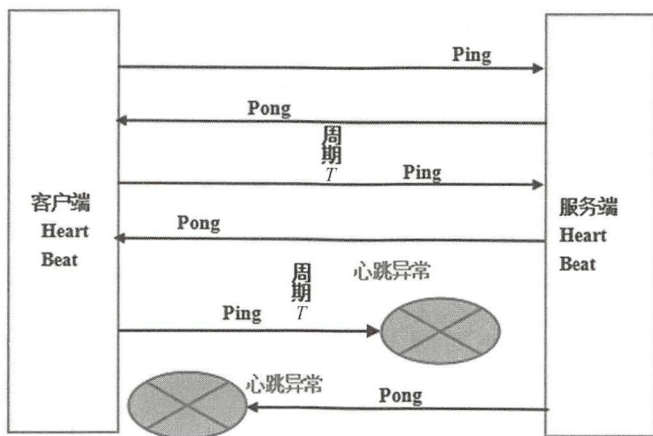


图 11-4 心跳检测机制的工作原理

心跳检测策略如下。

(1) 连续  $N$  次心跳检测都没有收到对方的 Pong 应答消息或者 Ping 请求消息，则认为链路已经发生逻辑失效，这被称为心跳超时。

(2) 在读取和发送心跳消息的时候如果直接发生了 I/O 异常，说明链路已经失效，这被称为心跳失败。无论发生心跳超时还是心跳失败，都需要关闭链路，由客户端发起重连操作，保证链路能够恢复正常。

Netty 提供了三种链路空闲检测机制，利用该机制可以轻松地完成心跳检测。

(1) 读空闲，链路持续时间  $T$  没有读取到任何消息。

(2) 写空闲，链路持续时间  $T$  没有发送任何消息。

(3) 读写空闲，链路持续时间  $T$  没有接收或者发送任何消息。

链路空闲事件被触发后并没有关闭链路，而是触发 `IdleStateEvent` 事件，用户订阅 `IdleStateEvent` 事件，用于自定义逻辑处理，例如关闭链路、客户端发起重新连接、告警和打印日志等。

利用 Netty 提供的链路空闲检测机制，可以非常灵活地实现协议层的心跳检测。如果选择双向心跳，在初始化 Channel 时将 Netty 的 `IdleStateHandler` 实例添加到 `ChannelPipeline` 中，然后监听 `READER_IDLE` 事件，一旦 `READER_IDLE` 事件发生，说明周期  $T$  内没有





读取到设备的消息，触发服务端主动发送心跳，检测链路是否存活，如果发生 I/O 异常说明链路已经失效，则主动关闭链路；如果发送成功，则等待最终的心跳超时，即在连续  $N$  个周期  $T$  内都没有接收到端侧设备发送的业务数据或者心跳消息，则说明端侧设备已经发生故障，服务端主动关闭连接，释放资源。采用双向心跳检测的主要优点有两个。

(1) 可以及时识别网络单通、对方突然掉电等特殊异常。

(2) 可以识别对方是否能够正常工作，而不仅是网络层面的互通性检测。

除了 `IdleStateHandler`，也可以根据实际需要选择 `ReadTimeoutHandler` 或者 `WriteTimeoutHandler`，链路空闲检测相关类库如图 11-5 所示。

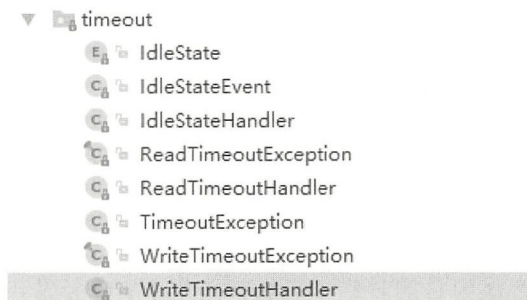


图 11-5 链路空闲检测相关类库

对于 IoT 场景，不建议过长的心跳检测周期和超时机制，主要有如下几点考虑。

(1) 百万级的长连接就有百万级的定时器，这么庞大的定时器会占用大量内存，如果长时间存活，会被“晋升”到老年代，加重 CMS 等老年代垃圾收集器的负担，容易导致 STW 问题。

(2) 过长的心跳检测超时不能及时发现掉线的设备（例如突然掉电），导致大量无效的 TCP 连接在内存中积压，同时占用操作系统句柄，影响性能，也容易导致 OOM 异常。

在创建 `IdleStateHandler` 实例时，可以指定空闲检测时间，代码如下（`IdleStateHandler` 类）：

```
public IdleStateHandler(boolean observeOutput,
                        long readerIdleTime, long writerIdleTime, long allIdleTime,
                        TimeUnit unit) {
    if (unit == null) {
```



```
        throw new NullPointerException("unit");
    }
    this.observeOutput = observeOutput;
    if (readerIdleTime <= 0) {
        readerIdleTimeNanos = 0;
    } else {
        readerIdleTimeNanos = Math.max(unit.toNanos(readerIdleTime),
MIN_TIMEOUT_NANOS);
    }
    if (writerIdleTime <= 0) {
        writerIdleTimeNanos = 0;
    } else {
        writerIdleTimeNanos = Math.max(unit.toNanos(writerIdleTime),
MIN_TIMEOUT_NANOS);
    }
    if (allIdleTime <= 0) {
        allIdleTimeNanos = 0;
    } else {
        allIdleTimeNanos = Math.max(unit.toNanos(allIdleTime),
MIN_TIMEOUT_NANOS);
    }
}
```

---

### 11.4.3 接收和发送缓冲区调优

在一些场景下，端侧设备会周期性地上报数据和发送心跳，单个链路的消息收发量并不大，针对此类场景，可以通过调小 TCP 的接收和发送缓冲区来降低单个 TCP 连接的资源占用率，例如将收发缓冲区设置为 8KB，相关代码如下：

---

```
//代码省略
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
```



```
        ChannelPipeline p = ch.pipeline();
        p.addLast(serviceHandler);
    }

    }).childOption(ChannelOption.SO_RCVBUF, 8 * 1024)
    .childOption(ChannelOption.SO_SNDBUF, 8 * 1024);
//代码省略
}
```

需要指出的是，对于不同的应用场景，收发缓冲区的最优值可能不同，用户需要根据实际场景，结合性能测试数据进行针对性的调优。

#### 11.4.4 合理使用内存池

随着 JVM 虚拟机和 JIT 即时编译技术的发展，对象的分配和回收是一个非常轻量级的工作。但是对于缓冲区 Buffer，情况却稍有不同，特别是堆外直接内存的分配和回收，是一个耗时的操作。为了尽量重用缓冲区，Netty 提供了基于内存池的缓冲区重用机制。

在物联网场景下，需要为每个接入的端侧设备至少分配一个接收和发送缓冲区对象，采用传统的非池模式，每次消息读写都需要创建和释放 ByteBuf 对象，如果有 100 万个连接，每秒上报一次数据或者心跳，就会有 100 万次/秒的 ByteBuf 对象申请和释放，即便服务端的内存可以满足要求，GC 的压力也会非常大。

以上问题最有效的解决方法就是使用内存池，每个 NioEventLoop 线程处理  $N$  个链路，在线程内部，链路的处理是串行的。假如 A 链路首先被处理，它会创建接收缓冲区等对象，待解码完成，构造的 POJO 对象被封装成任务后投递到后台的线程池中执行，然后接收缓冲区会被释放，每条消息的接收和处理都会重复接收缓冲区的创建和释放。如果使用内存池，则当 A 链路接收到新的数据报时，从 NioEventLoop 的内存池中申请空闲的 ByteBuf，解码后调用 release 将 ByteBuf 释放到内存池中，供后续的 B 链路使用。

Netty 内存池从实现上可以分为两类：堆外直接内存和堆内存。由于 ByteBuf 主要用于网络 I/O 读写，因此采用堆外直接内存会减少一次从用户堆内存到内核态的字节数组拷贝，所以性能更高。由于 DirectByteBuf 的创建成本比较高，因此如果使用 DirectByteBuf，则需要配合内存池使用，否则性价比可能还不如 HeapByteBuf。



Netty 默认的 I/O 读写操作采用的都是内存池的堆外直接内存模式，如果用户需要额外使用 ByteBuf，建议也采用内存池方式；如果不涉及网络 I/O 操作（只是纯粹的内存操作），可以使用堆内存池，这样内存的创建效率会更高一些。

由于堆外直接内存定位内存泄漏等问题不太方便，有时候需要在测试环境将内存分配策略调整为堆内存模式，待问题解决后再切换到堆外直接内存。假如将消息读取 ByteBuf 设置为非内存池、堆内存模式，代码示例如下：

```
//代码省略
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.config().setAllocator(UnpooledByteBufAllocator.DEFAULT);
        ChannelPipeline p = ch.pipeline();
//后续代码省略
}
```

设置 `Dio.netty.noUnsafe` 属性为 `true`，使用 `Heap` 堆内存方式创建 `ByteBuf`，启动参数设置如图 11-6 所示。



图 11-6 启动参数设置

修改后，在服务端消息接收处设置断点，发现 `ByteBuf` 为 `UnpooledHeapByteBuf`，调测结果如图 11-7 所示。

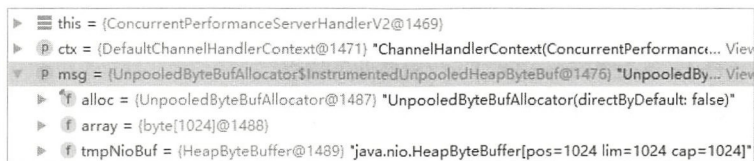


图 11-7 修改后测试结果

### 11.4.5 防止 I/O 线程被意外阻塞

通常情况下，大家都知道不能在 Netty 的 I/O 线程上做执行时间不可控的操作，例如访



问数据库、调用第三方服务等。但是有些隐形的阻塞操作却容易被忽略，例如打印日志。

在生产环境中，通常需要实时打印接口日志，其他日志处于 ERROR 级别，当服务发生 I/O 异常时，会记录异常日志。如果当前磁盘的 WIO 比较高，写日志文件操作可能会被同步阻塞（阻塞时间无法预测）。这就会导致 Netty 的 NioEventLoop 线程被阻塞，Socket 链路无法被及时关闭，其他的链路也无法进行读写操作。

以最常用的 log4j（1.2.X 版本）为例，尽管它支持异步写日志（AsyncAppender），但是当日志队列满时，它会同步阻塞业务线程（采用等待非丢弃方式时），直到日志队列有空闲位置可用，相关代码如下：

---

```
//代码省略
synchronized (this.buffer) {
    while (true) {
        int previousSize = this.buffer.size();
        if (previousSize < this.bufferSize) {
            this.buffer.add(event);
            if (previousSize != 0) break;
            this.buffer.notifyAll(); break;
        }
        boolean discard = true;
        if ((this.blocking) && (!Thread.interrupted()) &&
            (Thread.currentThread() != this.dispatcher)) //判断是不是业务线程
        {
            try
            {
                this.buffer.wait(); //阻塞业务线程
                discard = false;
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
        }
        //后续代码省略
    }
}
```

---





类似问题具有极强的隐蔽性，往往 WIO 高的时间持续非常短，或者是偶现的，在测试环境中很难模拟此类故障，问题定位难度非常大。一旦在生产环境中出现问题，在测试环境中又无法重现，就会比较被动。

### 11.4.6 I/O 线程和业务线程分离

如果服务端不做复杂的业务逻辑操作，仅是简单的内存操作和消息转发，则可以通过调大 NioEventLoop 工作线程池的方式，直接在 I/O 线程中执行业务 ChannelHandler，这样便减少了一次线程上下文切换，性能反而更高。

如果有复杂的业务逻辑操作，则建议 I/O 线程和业务线程分离，对于 I/O 线程，由于互相之间不存在锁竞争，可以创建一个大的 NioEventLoopGroup 线程组，所有 Channel 都共享同一个线程池。对于后端的业务线程池，则建议创建多个小的业务线程池，线程池可以与 I/O 线程绑定，这样既减少了锁竞争，又提升了后端的处理性能。I/O 线程和业务线程分离原理如图 11-8 所示。

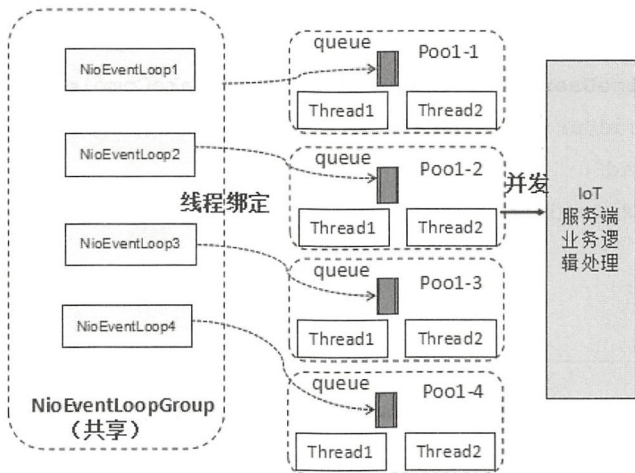


图 11-8 I/O 线程和业务线程分离原理

### 11.4.7 针对端侧并发连接数的流控

无论服务端性能优化到多少，都需要考虑流控功能。当资源成为瓶颈，或者遇到端



侧设备的大量接入，需要通过流控对系统做保护。流控的策略有很多种，IoT 场景最重要的就是针对端侧连接数的流控。

在 Netty 中，可以非常方便地实现流控功能：新增一个 `FlowControlChannelHandler`，添加到 `ChannelPipeline` 靠前的位置，继承 `channelActive()` 方法，创建 TCP 链路后，执行流控逻辑，如果达到流控阈值，则拒绝该连接，调用 `ChannelHandlerContext` 的 `close()` 方法关闭连接。

TLS/SSL 的连接数的流控相对复杂一些，可以在 TLS/SSL 握手成功后，监听握手成功的事件，执行流控逻辑。握手成功后发送 `SslHandshakeCompletionEvent` 事件，代码示例如下（`SslHandler` 类）：

---

```
private void setHandshakeSuccess() {
    handshakePromise.trySuccess(ctx.channel());
    if (logger.isDebugEnabled()) {
        logger.debug("{} HANDSHAKEN: {}", ctx.channel(),
engine.getSession().getCipherSuite());
    }
    ctx.fireUserEventTriggered(SslHandshakeCompletionEvent.SUCCESS);
    if (readDuringHandshake && !ctx.channel().config().isAutoRead()) {
        readDuringHandshake = false;
        ctx.read();
    }
    //后续代码省略
}
```

---

`FlowControlChannelHandler` 继承 `userEventTriggered()` 方法，拦截 TLS/SSL 握手成功事件，执行流控逻辑，示例代码如下：

---

```
public void userEventTriggered(ChannelHandlerContext ctx, Object evt)
throws Exception {
    if (evt == SslHandshakeCompletionEvent.SUCCESS) {
        //执行流控逻辑
    }
}
```

---



## 11.5 JVM 相关性能优化

JVM 层面的调优主要涉及 GC 参数优化,GC 参数设置不当会导致频繁 GC,甚至 OOM 异常,对服务端的稳定运行产生重大影响。

### 11.5.1 GC 调优

#### 1. 确定 GC 优化目标

GC (垃圾收集) 有三个主要指标。

(1) 吞吐量: 是评价 GC 能力的重要指标,在不考虑 GC 引起的停顿时间或内存消耗时,吞吐量是 GC 能支撑应用程序达到的最高性能指标。

(2) 延迟: GC 能力的最重要指标之一,是由于 GC 引起的停顿时间,优化目标是缩短延迟时间或完全消除停顿 (STW),避免应用程序在运行过程中发生抖动。

(3) 内存占用: GC 正常时占用的内存量。

JVM GC 调优的三个基本原则如下。

(1) Minor GC 回收原则: 每次新生代 GC 回收尽可能多的内存,减少应用程序发生 Full GC 的频率。

(2) GC 内存最大化原则: 垃圾收集器能够使用的内存越大,垃圾收集效率越高,应用程序运行也越流畅。但是过大的内存一次 Full GC 耗时可能较长,如果能够有效避免 Full GC,就需要做精细化调优。

(3) 3 选 2 原则: 吞吐量、延迟和内存占用不能兼得,无法同时做到吞吐量和暂停时间都最优,需要根据业务场景做选择。对于大多数 IoT 应用,吞吐量优先,其次是延迟。当然对于时延敏感型的业务,需要调整次序。

#### 2. 确定服务端内存占用

在优化 GC 之前,需要确定应用程序的内存占用大小,以便为应用程序设置合适的内存,提升 GC 效率。内存占用与活跃数据有关,活跃数据指的是应用程序稳定运行时长时

间存活的 Java 对象。活跃数据的计算方式：通过 GC 日志采集 GC 数据，获取应用程序稳定时老年代占用的 Java 堆大小，以及永久代（元数据区）占用的 Java 堆大小，两者之和就是活跃数据的内存占用大小。

### 3. GC 数据的采集

GC 数据的采集有两种方式。

(1) 通过 `-XX:+PrintGC`、`-XX:+PrintGCDetails` 和 `-XX:+PrintGCDateStamps` 等参数打印 GC 日志，通过 GC 日志采集数据如图 11-9 所示。

```
Java HotSpot(TM) 64-Bit Server VM (25.131-b11) for windows-amd64 JRE (1.8.0_
Memory: 4k page, physical 4007572k(814956k free), swap 9625496k(1918240k fre
CommandLine flags: -XX:InitialHeapSize=64121152 -XX:MaxHeapSize=67108864 -XX
2018-08-26T10:28:06.045+0800: 13.662: [GC (Allocation Failure)
[PSYoungGen: 15872K->2555K(18432K)] 15872K->4066K(60928K), 0.0344855 secs]
[Times: user=0.03 sys=0.00, real=0.03 secs]
2018-08-26T10:28:27.106+0800: 34.724: [GC (Allocation Failure) [PSYoungGen:
```

图 11-9 通过 GC 日志采集数据

(2) 通过 Visual GC 工具监控 GC 数据，如图 11-10 所示。

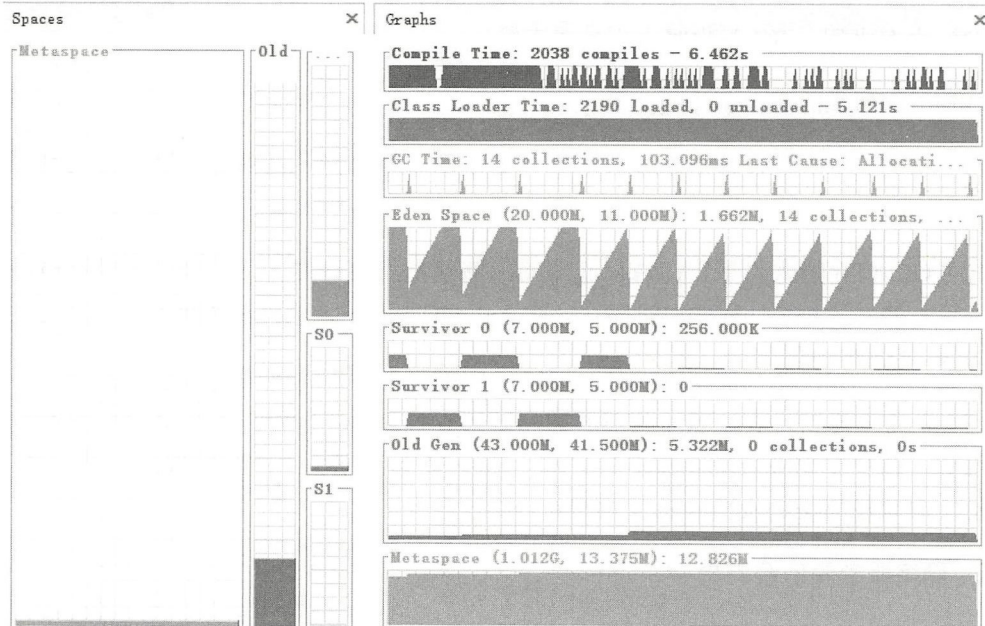


图 11-10 通过 GCVisual 工具监控 GC 数据



如果应用程序没有或者很少发生 Full GC，则可以通过 jvisualvm 等工具人工触发 Full GC，如图 11-11 所示。



图 11-11 人工触发 Full GC

执行强制 GC 之后，发现 GC 日志打印了 Full GC 相关数据，如图 11-12 所示。

```
2018-08-26T10:38:54.641+0800: 86.863: [GC (System.gc()) [PSYoungGen: 7776K->2312K(15360K)] 10758K->5294K(57856K)
2018-08-26T10:38:54.647+0800: 86.869: [Full GC (System.gc()) [PSYoungGen: 2312K->0K(15360K)]
[ParOldGen: 2982K->3122K(42496K)] 5294K->3122K(57856K), [Metaspace: 13084K->13018K(1060864K)], 0.0391629 secs]
[Times: user=0.08 sys=0.00, real=0.04 secs]
```

图 11-12 Full GC 相关数据

#### 4. GC 数据解读

通过打印 GC 日志方式来采集垃圾回收数据，需要人工解读 GC 相关信息，新生代 GC 数据解读如图 11-13 所示。

```
2018-08-26T10:37:39.738+0800: 11.960: [GC (Allocation Failure)
[PSYoungGen: 15872K->2552K(18432K)] ①
⑤15872K->4080K(60928K), 0.0257886 secs]
[Times: user=0.00 sys=0.00, real=0.03③ secs]
```

① 新生代内存分配失败，触发一次Young GC  
 ② 新生代GC之前内存占用  
 ③ 新生代GC之后内存占用  
 ④ 新生代总内存大小  
 ⑤ 新生代GC之前堆内存大小，可以看出老年代为空  
 ⑥ 新生代GC之后，整个堆内存使用的空间大小  
 ⑦ 整个堆内存占用总空间  
 ⑧ 本次新生代GC的总耗时30毫秒

图 11-13 新生代 GC 数据解读

#### 5. Java 堆大小设置原则（如表 11-1 所示）

表 11-1 Java 堆大小设置原则

空间	VM 命令行参数	占用比例
Java 堆	-Xms\ -Xmx	3~4 倍 Full GC 后的老年代空间占用
新生代	-Xmn	1~1.5 倍 Full GC 后的老年代空间占用
老年代	堆大小-新生代大小	2~3 倍 Full GC 后的老年代空间占用
永久代	-XX:PermSize/-XX: MaxPermSize	1.2~1.5 倍 Full GC 后的永久代空间占用



上述经验数据已经可以满足大多数应用场景的需要，如果业务场景比较特殊，可以根据 GC 统计数据和应用性能测试结果进行相应的优化。

## 6. 垃圾收集器的选择

如果是 JDK8 及以上版本，建议选择 G1 收集器，如果是较低版本的 JDK，或者业务已经积累了一些优化的 GC 参数，则可以继续使用“ParNew 收集器 + CMS 收集器”组合，根据测试情况和对应垃圾收集器的特点做相应的调优。

CMS 吞吐量调优的主要策略如下。

(1) 增加新生代空间，降低新生代 GC 频率，减少固定时间内新生代 GC 的次数。

(2) 增加老年代空间，降低 CMS 的频率并减少内存碎片，最终减小并发模式失效引起 Full GC 发生的概率。

(3) 调整新生代 Eden 和 Survivor 空间的大小比例，减少由新生代晋升到老年代的对象数目，降低 CMS GC 频率。

G1 调优的策略如下。

(1) 不要使用 -Xmn 选型或者 -XX:NewRatio 等其他相关选型显式设置年轻代的大小，这样会覆盖暂停时间指标。

(2) 暂停时间不要设置得太小，否则为了达到暂停时间目标会增加垃圾回收的开销，影响吞吐量指标。

(3) 防止触发 Full GC：在某些情况下，例如并发模式失败，G1 会触发 Full GC，这时 G1 会退化使用 Serial 收集器来完成垃圾清理工作，它仅使用单线程来完成 GC，GC 暂停时间可能会达到秒级。

## 7. 一些 GC 调优误区

在实际项目中，针对 GC 相关的一些认知或者做法存在误区，总结如下：

(1) 在生产环境中不配置 GC 日志打印参数，担心影响业务性能。

(2) GC 日志格式选择 -XX:+PrintGCTimeStamps，导致 GC 日志很难跟其他业务日志对应起来。

(3) GC 日志文件路径设置为静态路径, 例如 `gc.log`, 没有配置绕接、切换策略, 导致重启之后日志被覆盖。

(4) GC 日志文件没有配置单个文件大小、绕接和备份机制, 导致单个 GC 文件过大。

(5) 只有 Full GC 才会导致应用暂停, 分析 STW 问题时直接使用 Full GC 关键字搜索, 其他的不看。

(6) 给出最优的 GC 参数或者明确 GC 优化方向之后, 通过一次调整就能解决问题。

(7) 业务内存使用不当导致的性能问题, 希望通过 GC 参数优化解决问题, 业务不用改代码。

(8) 只有 GC 才会导致应用暂停。

## 11.5.2 其他优化手段

在实际测试时, 往往需要先在测试环境模拟海量端侧设备的接入, 为了方便测试, 节约机器资源, 需要在客户端配置虚拟 IP:

---

```
# ifconfig eth0:1 XX.XX.XX.XX netmask 255.255.255.0
```

---

配置完成后通过 `ifconfig eth0:1` 查看 IP 与子网掩码是否正确, 如果没有问题, 通过 `ifconfig eth0:1 up` 启动新增的虚拟 IP 进行测试。

操作系统对单个 IP 是有连接数限制的, 每个 IP 对应的端口范围为 0~65535, 其中 0~1023 被系统占用, 所以连接能够分配的端口从 1024 开始, 考虑到其他进程的端口占用, 单个 IP 能够接入的连接数约为 6 万个。如果系统支持多网卡, 则可以采用多网卡、多 IP 的方式解决, 否则需要使用虚拟 IP 的方式解决连接数限制问题。

## 11.6 总结

除了通过操作系统内核参数、Netty 框架和 JVM 调优来提升单节点处理性能, 还可以

通过分布式集群的方式提升整个服务端的处理能力，把性能的压力分散到各个节点上。除了可以降低单个节点的风险，也可以利用云平台的弹性伸缩实现服务端的快速扩容，以应对突发的流量洪峰。如果每个节点负担过重，一旦某个节点宕机，流量会瞬间转移到其他节点，导致其他节点超负荷运行，系统的可靠性降低。通过“分布式 + 弹性伸缩”构建可平滑扩容的 IoT 服务端，是未来的一种主流模式。



## 第 12 章

---

# 静态检查修改不当引起性能下降案例

将业务代码提交到配置库做持续集成构建时，通常会对代码做静态检查，例如 FindBugs、Checkstyle 等。很多项目对静态检查出的问题都有清零要求，除了编码和单元测试，修改静态检查问题也是开发人员的日常工作之一。

有些静态检查问题按照建议修改也许能够消除错误，但是放到整个业务上下文中就有问题，因此开发人员不能机械地以消除静态告警或错误为目标，而是要结合代码上下文和业务场景进行修改，防止因为修改不当引起性能问题，以及其他问题。

### 12.1 Edge Service 性能严重下降问题

---

对基于 Netty 4.1 构建的 Edge Service 新版本进行性能测试，Restful 透传场景，性能相比上一个版本下降超过了 50%，无法满足业务的性能需求，需要定位问题原因并优化平台性能。

#### 12.1.1 Edge Service 热点代码分析

---

在新版本性能测试时抓取热点代码（CPU 执行时间维度），如图 12-1 所示。





图 12-1 Edge Service 热点代码

通过方法栈调用分析，发现平台在获取 Restful 请求消息体时，调用了字节数组拷贝方法，成为性能热点和瓶颈点，相关示例代码如下（RestfulReq 类）：

```
public byte [] body() {
    if (this.body != null)
        return Arrays.copyOf(this.body, this.body.length);
    return null;
}
```

在上一个版本中，获取请求消息体时直接返回的就是 body 对象，没有额外做内存拷贝，新版本为什么要修改原有的实现方式呢？

查看 Git 提交记录，与修改人员进行交流后得知，这段代码的修改原因竟然是 CI 静态检查（FindBugs/PMD 等）报错，按照公司的要求，静态检查相关问题要控制在一定范围内，于是按照静态检查工具的建议进行了修改。

静态检查报错的原因：如果直接返回 body 对象，这个对象可能会被引用者修改，为了防止被非法或者意外修改，所以需要每次获取时拷贝一个新的字节数组。工具建议的修改策略如图 12-2 所示。





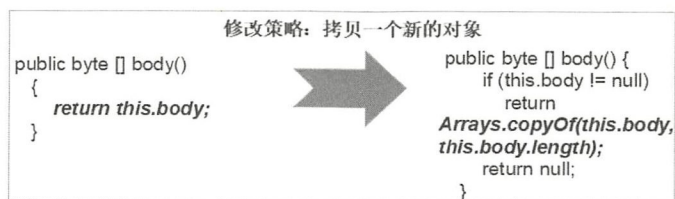


图 12-2 工具建议的修改策略

类似这样的修改，在整个项目中还存在多处，原因也基本雷同，当把这些所有类似修改组合在一起时，整个 Edge Service 的性能就下降了很多。

### 12.1.2 静态检查问题不是简单的一改了之

从表面上看，这个修改在功能上没有任何问题，功能测试用例也能通过，但是却存在如下几个隐患。

(1) 这是 Edge Service 平台的 Restful 协议栈，每次 API 调用都需要获取 body 并在应用层做反序列化。如果每次调用都额外拷贝一次，对性能的影响比较大。

(2) 字节数组的申请和释放相对比较耗时，需要避免频繁新建和销毁，但是上述修改却反其道而行之。

(3) 对于调用方而言，通过属性的 get 方法获取对象通常获得的都是引用，而非对象克隆或者拷贝。这个是编程惯例，但是上述修改打破了这个惯例，在底层做了隐性拷贝，对调用方不可见，这是非常糟糕的做法。

上述代码修改只考虑了消除静态检查错误，但是却没有结合 Restful 透传场景的关键路径调用及性能指标进行综合评估，最终导致了严重的性能问题。

对于 Java 而言，尽管 JIT 和 GC 随着 JDK 新版本的发布不断优化，但是频繁、大量地创建生命周期较短的字节数组对象仍然会给系统带来巨大的压力，GC 和 CPU 资源的额外损耗会带来性能问题。为了验证该问题，开发一个简单的测试用例（MockEdgeService 类）：

```
static void testCopyHotMethod() throws Exception
{
```



```
ByteBuf buf = Unpooled.buffer(1024);
for(int i = 0; i < 1024; i++)
    buf.writeByte(i);
RestfulReq req = new RestfulReq(buf.array());
while (true)
{
    byte [] msgReq = req.body();
}
}
```

其中，req.body()的实现如下：

```
public byte [] body() {
    if (this.body != null)
        return Arrays.copyOf(this.body, this.body.length);
    return null;
}
```

在性能测试几分钟之后，采集 GC 监控数据，发现发生了 7 万多次新生代 GC，GC 共耗时 49s，如图 12-3 所示。

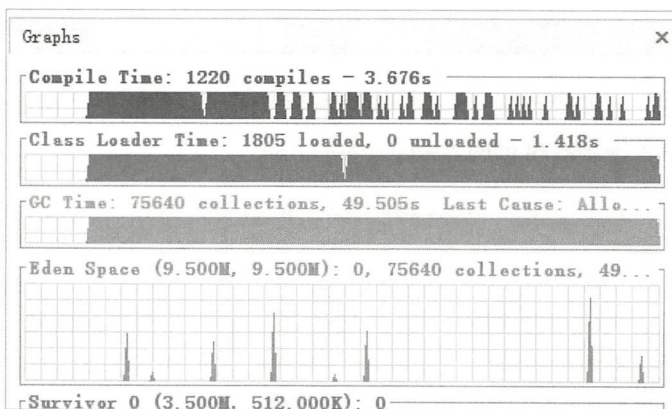


图 12-3 字节数组拷贝方式性能测试 GC 监控数据

采集 CPU 监控数据，发现 GC 占用 6%左右的 CPU 资源（应用整体占用 30%），即当前业务 20%左右的 CPU 资源被用于 GC，如图 12-4 所示。



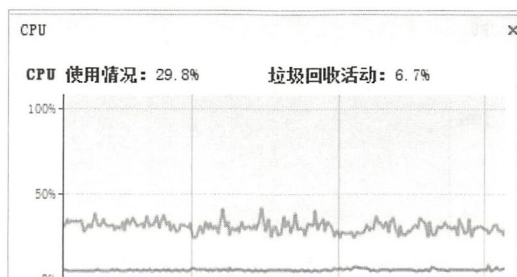


图 12-4 字节数组拷贝方式性能测试 CPU 监控数据

作为对比，将字节数组拷贝方式修改成直接获取请求 body 对象，代码如下（RestfulReqV2 类）：

---

```
public byte [] body()
{
    return this.body;
}
```

---

对 GC 数据做监控，发现系统只发生了 3 次新生代 GC，共耗时 12ms 左右，如图 12-5 所示。

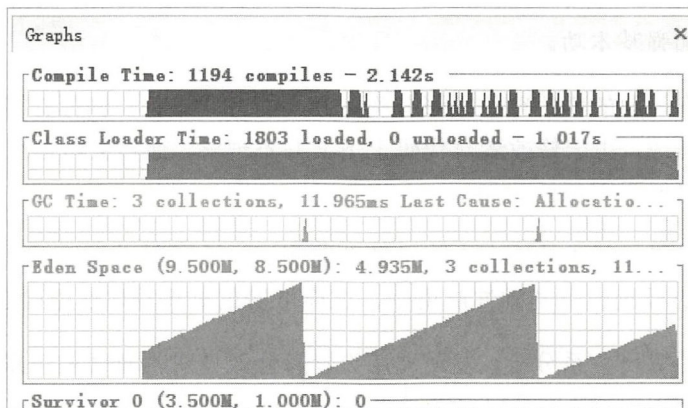


图 12-5 对象引用方式性能测试 GC 监控数据

查看 CPU 监控数据，发现 GC 使用的 CPU 为 0，说明系统当前 GC 工作正常，如图 12-6 所示。



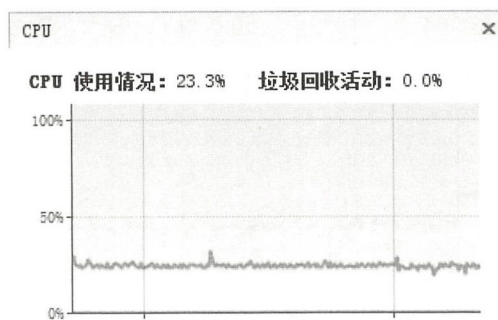


图 12-6 对象引用方式性能测试 CPU 监控数据

### 12.1.3 问题反思和改进

一个不经意的静态检查问题修改，却引发了严重的性能问题，在这次代码修改中，如下几个问题值得反思。

(1) 静态检查本来是用于解决问题的，却因为修改不当引入了新的问题。对于工具和规则不要盲从，需要结合业务场景辩证地看待，而不是为了完成任务和指标而盲目修改。

(2) 对于 Java 的克隆、对象拷贝等不太熟悉，也侧面反映出对 Java 基础知识掌握得不太扎实，需要加强基本功。

(3) 静态检查问题的修改往往按照分类、分人的方式进行，例如针对某一类问题，都统一由某一个人修改。由于修改者对代码上下文并不理解，通常会按照工具提示和建议进行修改，这样就可能引入新的问题。

(4) 对于关键路径代码的修改，除了做功能测试，还需要增加性能测试用例，例如每天 CI/CD 构建的时候跑一下基础性能用例，这样可以更早地发现性能问题。

通过对代码的引用关系分析，发现调用方没有修改请求消息体的需求，因此把对象拷贝修改为直接返回 body 对象引用。同时提供一个 copy 方法，用于调用方需要修改 body 的场景，通过名称调用方就知道自己做的是一次内存拷贝操作，而不是对象引用，代码示例如下（RestfulReqV2 类）：

---

```
public byte [] bodyCopy() {
    if (this.body != null)
```



```
        return Arrays.copyOf(this.body, this.body.length);  
    return null;  
}
```

---

## 12.2 克隆和浅拷贝

在 Java 中,经常需要进行对象拷贝,由于 JDK 的 Object 类提供了原生的 clone 方法,因此使用 JDK 的 clone 方法进行对象拷贝(也称克隆)是个不错的选择。尽管 clone 方法比较简单,但是如果实现不当,很容易因为浅拷贝而导致原始对象被非法修改。

### 12.2.1 浅拷贝存在的问题

浅拷贝是指拷贝对象时仅仅拷贝对象本身及它包含的基本变量,而不拷贝对象聚合的其他引用对象,浅拷贝实现克隆示例如下。

UserInfo 类:

---

```
public class UserInfo implements Cloneable{  
    private int age;  
    private Address address;  
    public Object clone() throws CloneNotSupportedException  
    {  
        return super.clone();  
    }  
}
```

---

Address 类:

---

```
public class Address implements Cloneable{  
    private String city;  
    public Object clone() throws CloneNotSupportedException  
    {
```





```

        return super.clone();
    }
}

```

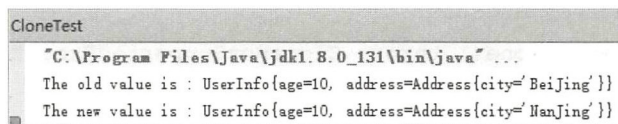
CloneTest 类:

```

static void testClone() throws Exception
{
    UserInfo user = new UserInfo();
    user.setAge(10);
    Address address = new Address();
    address.setCity("BeiJing");
    user.setAddress(address);
    System.out.println("The old value is : " + user);
    UserInfo cloneUser = (UserInfo)user.clone();
    cloneUser.setAge(20);
    cloneUser.getAddress().setCity("NanJing");
    System.out.println("The new value is : " + user);
}

```

克隆之后对新对象进行修改，分别修改基本类型的 age 和引用类型的 address，发现原始对象的 address 被修改，但是 age 没有被修改，如图 12-7 所示。



```

"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...
The old value is : UserInfo{age=10, address=Address{city='BeiJing'}}
The new value is : UserInfo{age=10, address=Address{city='NanJing'}}

```

图 12-7 浅拷贝实现的克隆测试结果

在大部分场景下，克隆的目的就是修改克隆对象后不影响原对象，浅拷贝实现的克隆显然不满足业务要求。如果要想实现深拷贝，需要对上述的 clone 方法进行修改，对自身包含的引用对象进行克隆，然后将克隆后的引用对象设置到自身的克隆对象中，示例代码如下（UserInfoV2 类）：

```

public Object clone() throws CloneNotSupportedException
{

```



```

UserInfoV2 cloneInfo = (UserInfoV2) super.clone();
cloneInfo.setAddress((Address) address.clone());
return cloneInfo;
}

```

对修改进行测试，发现尽管克隆之后的对象修改了引用对象的 address，但是并不影响原始对象的 address，如图 12-8 所示。

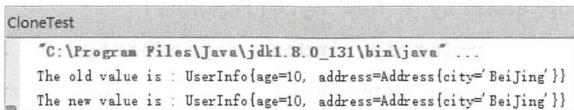


图 12-8 深拷贝实现的克隆测试结果

## 12.2.2 Netty 的对象拷贝实现策略

Netty 中涉及对象拷贝的主要是 ByteBuf 对象，ByteBuf 提供了多个接口用于创建某个 ByteBuf 的视图或者复制 ByteBuf，以满足用户的不同业务场景需要。具体策略如下。

(1) duplicate: 返回当前 ByteBuf 的拷贝对象，拷贝后返回的 ByteBuf 与操作的 ByteBuf 共享缓冲区内容，但是维护自己独立的读写索引。当修改拷贝后的 ByteBuf 内容时，之前原 ByteBuf 的内容也随之改变，双方持有的是同一个内容指针引用。

(2) copy: 拷贝一个新的 ByteBuf 对象，它的内容和索引都是独立的，拷贝操作本身并不修改原 ByteBuf 的读写索引。

(3) copy(int index, int length): 从指定的索引开始拷贝，拷贝的字节长度为 length，拷贝后的 ByteBuf 内容和读写索引都与之前的独立。

(4) slice: 返回当前 ByteBuf 的可读子缓冲区，起始位置从 readerIndex 到 writerIndex，返回的 ByteBuf 与原 ByteBuf 共享内容，但是读写索引独立维护。该操作并不修改原 ByteBuf 的 readerIndex 和 writerIndex。

(5) slice(int index, int length): 返回当前 ByteBuf 的可读子缓冲区，起始位置从 index 到 “index + length”，返回后的 ByteBuf 与原 ByteBuf 共享内容，但是读写索引独立维护。该操作并不修改原 ByteBuf 的 readerIndex 和 writerIndex。



以 `UnpooledHeapByteBuf` 为例，调用它的 `copy` 方法，会新建一个基于堆内存的 `byte` 数组，通过 `System.arraycopy` 将原始的 `ByteBuf` 内容拷贝到新创建的 `UnpooledHeapByteBuf` 中，新建 `UnpooledHeapByteBuf` 的 `readerIndex` 和 `writerIndex` 及内容都是独立的，与原始对象无关，它的代码实现如下（`UnpooledHeapByteBuf` 类）：

---

```
public ByteBuf copy(int index, int length) {
    checkIndex(index, length);
    byte[] copiedArray = new byte[length];
    System.arraycopy(array, index, copiedArray, 0, length);
    return new UnpooledHeapByteBuf(alloc(), copiedArray, maxCapacity());
}
```

---

## 12.3 总结

静态检查本来是为了提升代码质量，但是由于盲目按照工具的建议做修改，对业务运行态的关键代码路径及上下文场景都不清楚，最终导致了严重的性能问题。由于 Netty 通常被用于高性能的通信框架，所以任何涉及性能的修改一定要谨慎，修改之后需要结合业务场景做相应的性能测试，以验证修改是否合理。



## 第 13 章

---

# Netty 性能统计误区案例

通常情况下，用户以黑盒的方式使用 Netty，通过 Netty 完成协议消息的读取和发送，以及编解码操作，不需要关注 Netty 的底层实现细节。

在高并发场景下，往往需要统计系统的关键性能 KPI 数据，结合日志、告警等对故障进行定位分析，如果对 Netty 的底层实现细节不了解，获取哪些关键性能数据及数据正确的获取方式都将成为难点。错误或者不准确的数据可能影响问题定位的思路和方向，导致问题迟迟得不到正确的解决。

## 13.1 时延毛刺排查相关问题

---

某电商生产环境在业务高峰期，偶现服务调用时延毛刺问题，时延突然增大的服务没有固定规律，问题发生的比例虽然很低，但是对客户的体验影响很大，需要尽快定位问题原因并解决问题。

### 13.1.1 时延毛刺问题初步分析

---

服务调用时延增大，但并不是异常，因此运行日志并不会打印错误（ERROR）日志，



单靠传统的日志无法进行有效的问题定位。利用分布式消息跟踪系统，进行分布式环境的问题定位。

通过对服务调用时延进行排序和过滤，找出时延增大的服务调用链详细信息，发现业务服务端处理很快，但是消费者统计数据却显示服务端处理非常慢，调用链两端看到的数据不一致，怎么回事？

对调用链的详情进行分析，发现服务端打印的时延是业务服务接口调用的耗时，并不包含：

- （1）服务端读取请求消息、解码消息，以及内部消息投递、在线程池消息队列排队等待的时间。
- （2）响应消息编码时间、消息队列发送排队时间及消息写入 Socket 发送缓冲区的时间。

服务调用链的工作原理如图 13-1 所示。

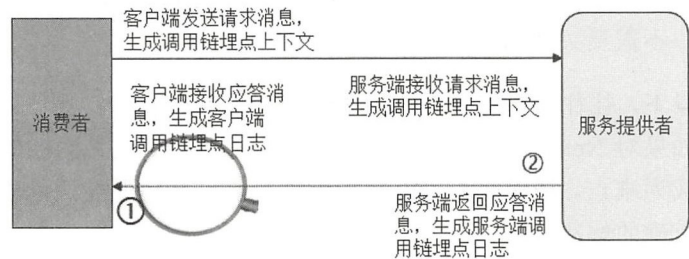


图 13-1 服务调用链的工作原理

将调用链中的消息调用过程详细展开，以服务端读取请求和发送响应消息为例进行说明，如图 13-2 所示。

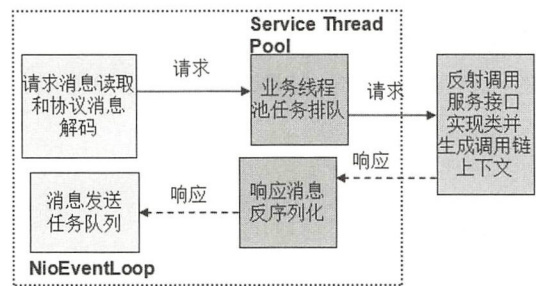


图 13-2 服务端调用链详情





服务端的处理耗时，除了业务服务自身调用的耗时，还应该包含服务框架的处理时间，具体如下。

- (1) 请求消息的解码（反序列化）时间。
- (2) 请求消息在业务线程池中排队等待执行时间。
- (3) 响应消息编码（序列化）时间。
- (4) 响应消息 ByteBuf 在发送队列中的排队时间。

由于服务端调用链只采集了业务服务接口的调用耗时，没有包含服务框架本身的调度和处理时间，导致无法对问题进行定位。服务端没有统计服务框架的处理时间，因此不排除消息发送队列或者业务线程池队列积压而导致时延变大。

### 13.1.2 服务调用链改进

对服务调用链埋点进行优化，具体措施如下。

- (1) 包含客户端和服务端消息编码和解码的耗时。
- (2) 包含请求和应答消息在队列中的排队时间。
- (3) 包含应答消息在通信线程发送队列（数组）中的排队时间。

同时，为了方便定位问题，增加打印输出 Netty 的性能统计日志，主要如下。

- (1) 当前系统的总链路数，以及每个链路的状态。
- (2) 每条链路接收的总字节数、周期  $T$  接收的字节数、消息接收吞吐量。
- (3) 每条链路发送的总字节数、周期  $T$  发送的字节数、消息发送吞吐量。

优化服务调用链之后，上线运行一段时间，通过分析 Netty 性能统计日志、调用链日志，发现双方的数据并不一致，Netty 性能统计日志统计的数据与前端门户看到的也不一致，因此怀疑新增的性能统计功能存在缺陷，需要继续对问题进行定位。



### 13.1.3 都是同步思维惹的祸

传统的同步服务调用，发起服务调用之后，业务线程阻塞，等待响应，接收响应之后，业务线程继续执行，对发送的消息进行累加，获取性能 KPI 数据。

使用 Netty，所有的网络 I/O 操作都是异步执行的，即调用 Channel 的 write 方法，并不代表消息真正发送到 TCP 缓冲区中，如果在调用 write 方法之后就对发送的字节数做计数，统计结果就不准确。

对消息发送功能进行代码检查，发现代码调用 writeAndFlush 方法后直接对发送的请求消息字节数进行计数，代码如下（ServiceTraceServerHandler 类）：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    int sendBytes = ((ByteBuf)msg).readableBytes();
    ctx.writeAndFlush(msg);
    totalSendBytes.getAndAdd(sendBytes);
}
```

调用 writeAndFlush 并不代表消息已经发送到网络上，它仅仅是一个异步的消息发送操作，调用 writeAndFlush 之后，Netty 会执行一系列操作，最终将消息发送到网络上，writeAndFlush 处理流程如图 13-3 所示。

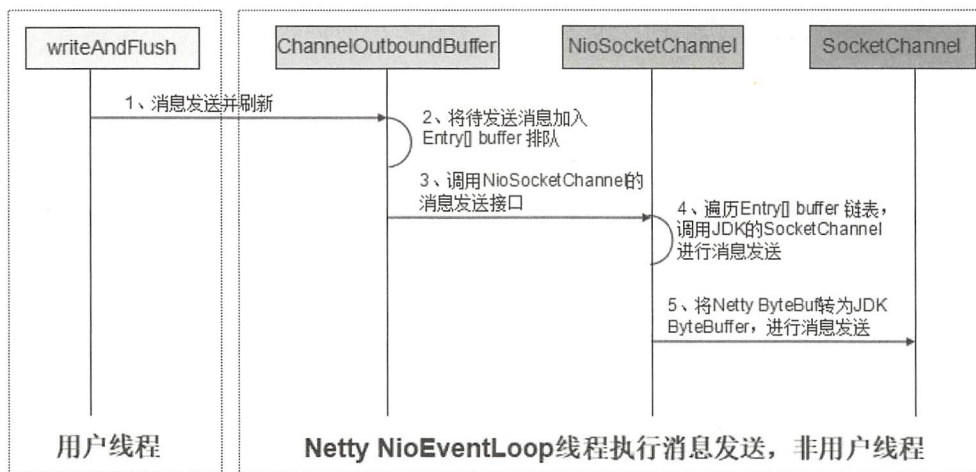


图 13-3 writeAndFlush 处理流程

通过深入分析 `writeAndFlush` 方法，我们发现性能统计代码忽略了如下几个耗时。

- (1) 业务 `ChannelHandler` 的执行时间。
- (2) 被异步封装的 `WriteTask/WriteAndFlushTask` 在 `NioEventLoop` 任务队列中的排队时间。
- (3) `ByteBuf` 在 `ChannelOutboundBuffer` 队列中的排队时间。
- (4) JDK NIO 类库将 `ByteBuffer` 写入网络的时间。

由于性能统计遗漏了上述 4 个关键步骤的执行时间，因此统计出来的发送速度比实际值会高一些，这将干扰我们的问题定位思路。

### 13.1.4 正确的消息发送速度性能统计策略

正确的消息发送速度性能统计策略如下。

- (1) 调用 `writeAndFlush` 方法之后获取 `ChannelFuture`。
- (2) 新增消息发送 `ChannelFutureListener` 并注册到 `ChannelFuture`，监听消息发送结果，如果消息写入 `SocketChannel` 成功，则 Netty 会回调 `ChannelFutureListener` 的 `operationComplete` 方法。
- (3) 在消息发送 `ChannelFutureListener` 的 `operationComplete` 方法进行性能统计。

正确的性能统计代码示例如下（`ServiceTraceServerHandlerV2` 类）：

---

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    int sendBytes = ((ByteBuf)msg).readableBytes();
    ChannelFuture writeFuture = ctx.write(msg);
    writeFuture.addListener((f) ->
    {
        totalSendBytes.getAndAdd(sendBytes);
    });
    ctx.flush();
}
```

---

对 Netty 消息发送相关源码进行分析，当发送的字节数大于 0 时，进行 ByteBuf 的清理工作，代码如下（NioSocketChannel 类）：

---

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    //代码省略
    if (localWrittenBytes <= 0) {
        incompleteWrite(true);
        return;
    }
    adjustMaxBytesPerGatheringWrite(attemptedBytes,
localWrittenBytes, maxBytesPerGatheringWrite);
    in.removeBytes(localWrittenBytes);
    --writeSpinCount;
    break;

    //代码省略
}
```

---

接着分析 ChannelOutboundBuffer 的 removeBytes(long writtenBytes)方法，将发送的字节数与当前 ByteBuf 可读的字节数进行对比，判断当前的 ByteBuf 是否完成发送，如果完成则调用 remove 方法清理它，否则只更新发送进度，相关代码如下（ChannelOutboundBuffer 类 removeBytes(long writtenBytes)方法）：

---

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    //代码省略
    if (readableBytes <= writtenBytes) {
        if (writtenBytes != 0) {
            progress(readableBytes);
            writtenBytes -= readableBytes;
        }
        remove();
    } else {
        if (writtenBytes != 0) {
            buf.readerIndex(readerIndex + (int) writtenBytes);
            progress(writtenBytes);
        }
    }
}
```

---

```

    }
    break;
}
//代码省略
}

```

当调用 `remove` 方法时，最终会调用消息发送 `ChannelPromise` 的 `trySuccess` 方法，通知监听消息已经完成发送，相关代码如下（`ChannelPromise` 类）：

```

public boolean trySuccess(V result) {
    //代码省略
    if (setSuccess0(result)) {
        notifyListeners();
        return true;
    }
    return false;
}
//代码省略
}

```

经过分析可以看出，调用 `write/writeAndFlush` 方法本身并不代表消息已经发送完成，只有监听 `write/writeAndFlush` 的操作结果，在异步回调监听中计数，结果才更精确。

需要注意的是，异步回调通知由 Netty 的 `NioEventLoop` 线程执行，即便异步回调代码写在业务线程中，也是由 Netty 的 I/O 线程来累加计数的，因此这里需要考虑多线程并发安全问题，如图 13-4 所示。



图 13-4 消息发送结果异步回调通知线程堆栈



如果消息报文比较大，或者一次批量发送的消息比较多，可能会出现“写半包”问题，即一个消息无法在一次 write 操作中全部完成发送，可能只发送了一半，针对此类场景，可以创建 GenericProgressiveFutureListener 用于实时监听消息发送进度，做更精准的统计，相关代码如下（DefaultPromise 类）：

---

```
private static void notifyProgressiveListeners0(
    ProgressiveFuture<?> future, GenericProgressiveFutureListener<?>[]
listeners, long progress, long total) {
    for (GenericProgressiveFutureListener<?> l: listeners) {
        if (l == null) {
            break;
        }
        notifyProgressiveListener0(future, l, progress, total);
    }
}
```

---

定位问题后，按照正确的做法对 Netty 性能统计代码进行修正，上线之后，结合调用链日志，很快定位出了业务高峰期偶现的部分服务时延毛刺较大问题，优化业务线程池参数配置，问题得到解决。

### 13.1.5 常见的消息发送性能统计误区

在实际业务中比较常见的性能统计误区如下。

（1）调用 write/writeAndFlush 方法就开始统计发送速度。

（2）消息编码时进行性能统计：编码之后，获取 out 可读的字节数，然后累加。编码完成并不代表消息被写入 SocketChannel，因此性能统计也不准确。

## 13.2 Netty 关键性能指标采集策略

除了消息发送速度，还有其他一些重要的指标需要采集和监控，无论是在调用链详情

中展示，还是统一由运维采集、汇总和展示，这些性能指标对于问题的定位帮助都很大。

### 13.2.1 Netty I/O 线程池性能指标

Netty I/O 线程池除了负责网络 I/O 消息的读写，还需要同时处理普通任务和定时任务，因此消息队列积压的任务个数是衡量 Netty I/O 线程池工作负载的重要指标。由于 Netty NIO 线程池采用的是一个线程池/组包含多个单线程线程池的机制，所以不需要像原生的 JDK 线程池那样统计工作线程数、最大线程数等指标，相关代码如下（ServiceTraceServerHandlerV2 类）：

---

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    kpiExecutorService.scheduleAtFixedRate(() ->
    {
        Iterator<EventExecutor> executorGroups = ctx.executor().
parent().iterator();
        while (executorGroups.hasNext())
        {
            SingleThreadEventExecutor executor =
(SingleThreadEventExecutor)executorGroups.next();
            int size = executor.pendingTasks();
            if (executor == ctx.executor())
                System.out.println(ctx.channel() + "--> " + executor + "
pending size in queue is : --> " + size);
            else
                System.out.println(executor + " pending size in queue is :
--> " + size);
        }
    }, 0, 1000, TimeUnit.MILLISECONDS);
}
```

---

Netty I/O 线程池性能统计程序运行结果如图 13-5 所示。

```
The server write rate is : 25600 bytes/s
[id: 0x83d86684, L:/127.0.0.1:18089 - R:/127.0.0.1:60642]--> io.netty.channel.nio.NioEventLoop@d70c109 pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@17ed40e0 pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@50675690 pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@31b7dea0 pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@3ac42916 pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@47d384ee pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@2d6a9952 pending size in queue is : --> 0
io.netty.channel.nio.NioEventLoop@22a71081 pending size in queue is : --> 0
```

图 13-5 Netty I/O 线程池性能统计程序运行结果

### 13.2.2 Netty 发送队列积压消息数

Netty 发送队列积压消息数可以反映网络速度、通信对端的读取速度及自身的发送速度等，因此对服务调用时延的精细化分析对于问题的定位非常有帮助，它的采集方式代码示例如下（ServiceTraceServerHandlerV2 类）：

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    writeQueKpiExecutorService.scheduleAtFixedRate(()->
    {
        long pendingSize = ((NioSocketChannel)ctx.channel()).unsafe().
outboundBuffer().totalPendingWriteBytes();
        System.out.println(ctx.channel() + "--> " +
"ChannelOutboundBuffer's totalPendingWriteBytes is : "
            + pendingSize + " bytes");
    },0,1000, TimeUnit.MILLISECONDS);
}
```

Netty 发送队列积压消息数代码运行结果如图 13-6 所示。

```
The server write rate is : 25600 bytes/s
[id: 0xc0ac61e0, L:/127.0.0.1:18089 - R:/127.0.0.1:61431]--> ChannelOutboundBuffer's totalPendingWriteBytes is : 0 bytes
[id: 0xc0ac61e0, L:/127.0.0.1:18089 - R:/127.0.0.1:61431]--> io.netty.channel.nio.NioEventLoop@d70c109 pending size in queue is : --> 0
```

图 13-6 Netty 发送队列积压消息数代码运行结果

由于 totalPendingSize 是 volatile 类型的，因此统计线程即便不是 Netty 的 I/O 线程，也能够正确地读取其最新值。

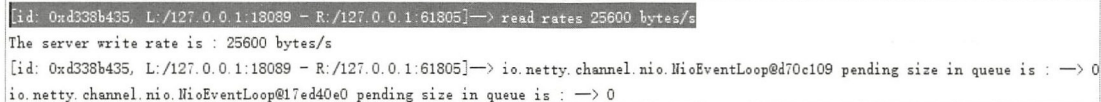
### 13.2.3 Netty 消息读取速度性能统计

针对某个 Channel 的消息读取速度性能统计,可以在解码 ChannelHandler 之前添加一个性能统计 ChannelHandler,用来对读取速度进行计数,相关代码示例如下 (ServiceTraceProfileServerHandler 类):

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    kpiExecutorService.scheduleAtFixedRate(()->
    {
        int readRates = totalReadBytes.getAndSet(0);
        System.out.println(ctx.channel() + "--> read rates " + readRates);
    },0,1000, TimeUnit.MILLISECONDS);
    ctx.fireChannelActive();
}

public void channelRead(ChannelHandlerContext ctx, Object msg) {
    int readableBytes = ((ByteBuf)msg).readableBytes();
    totalReadBytes.getAndAdd(readableBytes);
    ctx.fireChannelRead(msg);
}
```

Netty 消息读取速度性能统计代码运行结果如图 13-7 所示。



```
[id: 0xd338b435, L:/127.0.0.1:18089 - R:/127.0.0.1:61805]--> read rates 25600 bytes/s
The server write rate is : 25600 bytes/s
[id: 0xd338b435, L:/127.0.0.1:18089 - R:/127.0.0.1:61805]--> io.netty.channel.nio.NioEventLoop@470c109 pending size in queue is : -> 0
io.netty.channel.nio.NioEventLoop@17ed40e0 pending size in queue is : -> 0
```

图 13-7 Netty 消息读取速度性能统计代码运行结果

## 13.3 总结

当我们对服务调用时延进行精细化分析时,需要把 Netty 通信框架底层的处理耗时数据也采集并进行分析,由于 Netty 的 I/O 操作都是异步的,因此不能像传统同步调用那样做性能数据统计,需要注册性能统计监听器,在异步回调中完成计数。另外,Netty 的 I/O 线程池、消息发送队列等实现比较特殊,与传统的 Tomcat 等框架实现策略不同,因此 Netty 的关键性能数据采集不能照搬 JDK 和 Tomcat 的做法。



## 第 14 章

---

# gRPC 的 Netty HTTP/2 实践案例

gRPC 是由 Google 开发并开源的一种语言中立的 RPC 框架，当前支持 C、Java 和 Go 语言等，它是一个高性能的 RPC 框架，面向服务端和移动端，基于 HTTP/2 设计。

gRPC Java 版底层的通信框架基于 Netty 4.1 构建，通过集成 Netty 的 HTTP/2 协议栈，支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等特性，这些特性使得 gRPC 在移动设备上更加省电和节省网络流量。

### 14.1 gRPC 基础入门

---

gRPC 是一个支持多语言的 RPC 框架，其中 Java 版采用 “Netty + Protocol buffer” 构建，除了 Protocol buffer 序列化机制，也支持 JSON 序列化机制。

#### 14.1.1 RPC 框架简介

---

RPC 框架的目标就是让远程服务调用更加简单、透明，RPC 框架负责屏蔽底层的传输





方式（TCP 或者 UDP）、序列化方式（XML/JSON/二进制）和通信细节。服务调用者可以像调用本地接口一样调用远程的服务提供者，而不需要关心底层通信细节和调用过程。

RPC 框架的调用原理如图 14-1 所示。

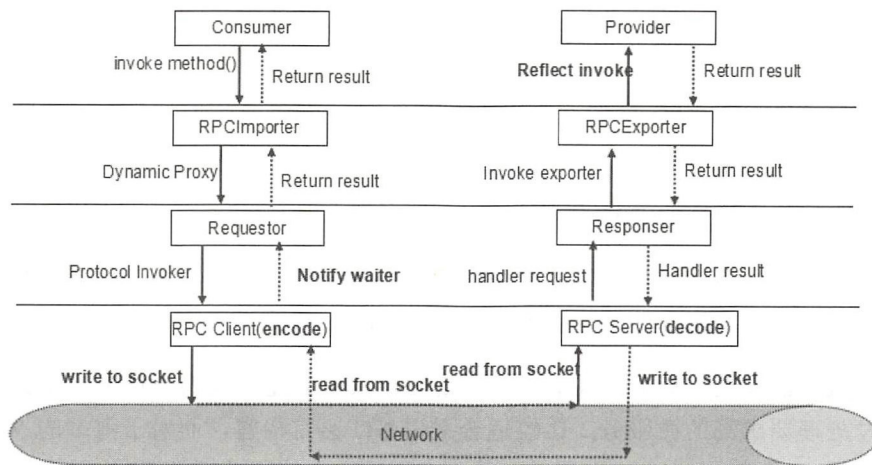


图 14-1 RPC 框架的调用原理

RPC 框架的几个核心技术点总结如下。

(1) 远程服务提供者需要以某种形式提供服务调用相关的信息，包括但不限于服务接口定义、数据结构，以及中间态的服务定义文件，例如 gRPC 的 proto 文件、WS-RPC 的 WSDL 文件定义，甚至也可以是服务端的接口说明文档。服务调用者需要通过一定的途径获取远程服务调用相关信息，例如服务端接口定义 Jar 包导入、获取服务端 IDL 文件等。

(2) 远程代理对象：服务调用者调用的服务实际是远程服务的本地代理，对于 Java 语言，它的实现就是 JDK 的动态代理，通过动态代理的拦截机制，将本地调用封装成远程服务调用。

(3) 通信：RPC 框架与具体的协议无关，例如 Spring 的远程调用支持 HTTP Invoke、RMI Invoke，MessagePack 使用的是私有的二进制压缩协议。

(4) 序列化：远程通信需要将对象转换成二进制码流进行网络传输，不同的序列化框架支持的数据类型、数据包大小、异常类型及性能等都不同。不同的 RPC 框架的应用场景不同，因此技术选择也存在很大差异。一些做得比较好的 RPC 框架，支持多种序列化



方式，有的甚至支持用户自定义序列化框架（Hadoop Avro）。

### 14.1.2 当前主流的 RPC 框架

当前主流的 RPC 框架整体上分为三类。

（1）支持多语言的 RPC 框架，比较成熟的有 Google 的 gRPC、Apache（Facebook）的 Thrift。

（2）只支持特定语言的 RPC 框架。

（3）支持服务治理等服务化特性的分布式服务框架，其底层内核仍然是 RPC 框架，例如华为开源的支持多语言的微服务框架 ServiceComb。

随着微服务的发展，基于语言中立性原则构建微服务逐渐成为一种主流模式，例如对于后端并发处理要求高的微服务，比较适合采用 GO 语言构建，而对于前端的 Web 界面，则更适合采用 Java 和 JavaScript。因此，基于多语言、支持移动端的 RPC 框架来构建微服务，是一种比较好的技术选择。

### 14.1.3 gRPC 框架特点

相比其他开源的 RPC 框架，gRPC 有如下几个特点。

（1）语言中立，支持多种语言。

（2）基于 IDL 文件定义服务，通过 proto3 工具生成指定语言的数据结构、服务端接口及客户端 Stub。

（3）通信协议基于标准的 HTTP/2 设计，支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等特性，这些特性使得 gRPC 在移动设备上更加省电和节省网络流量。

（4）序列化支持 Protocol buffer 和 JSON，Protocol buffer 是一种语言无关的高性能序列化框架，基于“HTTP/2 + Protocol buffer”，保障了 RPC 调用的高性能。



### 14.1.4 为什么选择 HTTP/2

传统的 HTTP/1.0 或者 HTTP/1.1（非 WebSocket 等）是无状态的，创建 HTTP 连接之后，客户端发送请求消息，然后等待服务端响应，接收到服务端响应之后，客户端接着发送后续的请求消息，服务端再返回响应，周而复始。请求和响应消息都是成对出现的，采用的是“一请求对应一响应”模式。在某个时刻，一个 HTTP 连接上只能单向地处理一个消息，就像单行道，一个消息处理得慢，很容易导致后续消息阻塞，它的原理如图 14-2 所示。

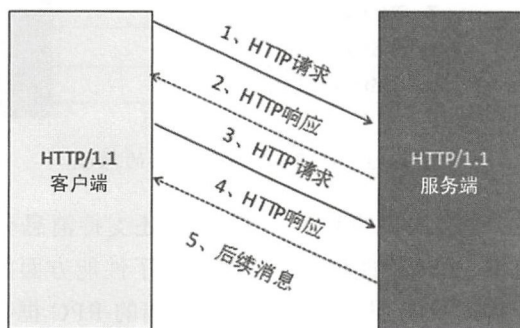


图 14-2 HTTP/1.1 消息交互的原理

采用该模式主要存在如下几个缺点。

- (1) 单个连接的通信效率不高，无法多路复用。
- (2) 一个请求消息处理得慢，很容易阻塞后续其他请求消息。

(3) 如果客户端读取响应超时，由于消息是无状态的，只能关闭连接，重建连接之后再发送请求。如果频繁地发生客户端超时，就会发生大量的 HTTP 连接断连和重连，如果采用 HTTPS，SSL 链路的重建成本很高，很容易导致服务端因负载过重而宕机。

(4) 为了解决单个 HTTP 连接性能不足问题，只能创建一个大的连接池，在大规模集群组网场景下，HTTP 连接数会非常多，额外占用大量句柄资源。

采用 HTTP/2 之后，可以实现多路复用，客户端可以连续发送多个请求，服务端也可以返回一个或者多个响应，而且还可以主动推送数据到服务端，实现双向通信，达到 TCP 私有协议的通信效果，它的原理如图 14-3 所示。

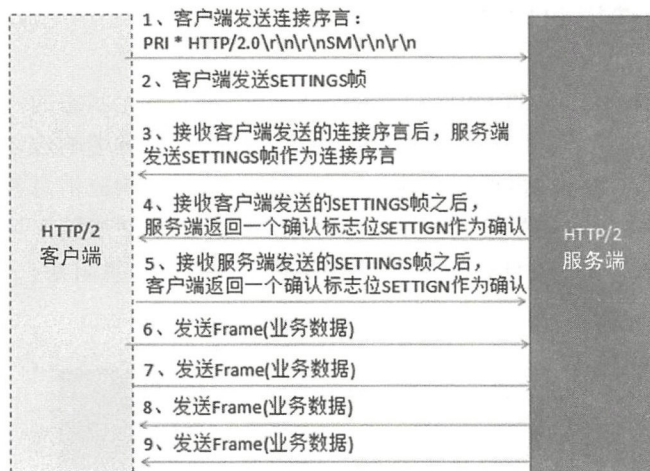


图 14-3 HTTP/2 消息交互的原理

除了可以实现 HTTP 连接的多路复用, HTTP/2 还支持消息头压缩、服务端推送等功能, 相比传统的 HTTP/1.1, 整体性能提升了很多。除了性能方面的优势, 越来越多的安卓 App 支持 HTTP/2, 对于同时支持服务端和安卓移动端的 RPC 框架而言, 采用 HTTP/2 是最佳选择。

## 14.2 gRPC Netty HTTP/2 服务端工作机制

gRPC 通过对 Netty HTTP/2 的封装, 向用户屏蔽底层 RPC 通信的协议细节。

### 14.2.1 Netty HTTP/2 服务端创建原理和源码分析

Netty HTTP/2 服务端创建的主要流程如下。

(1) 创建 NettyServerBuilder, 通过 Build 模式创建 NettyServer, 由 NettyServer 具体管理 HTTP/2 协议栈的启动和停止, 以及 HTTP/2 协议相关的各种参数。NettyServer 相关接口定义如图 14-4 所示。



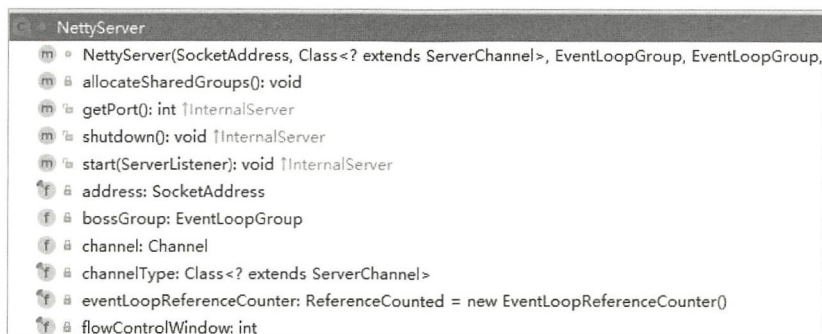


图 14-4 NettyServer 相关接口定义

(2) NettyServer 的 start 方法会创建 NettyServerTransport, 通过 NettyServerTransport 来创建 gRPC 的 Netty HTTP/2 ChannelHandler 实例, 并加入 ChannelPipeline。相关代码如下 (NettyServer 类):

```
public void start(ServerTransportListener listener) {
    Preconditions.checkNotNull(this.listener == null, "Handler already registered");
    this.listener = listener;

    final NettyServerHandler grpcHandler = createHandler(listener);
    HandlerSettings.setAutoWindow(grpcHandler);
    channel.closeFuture().addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            notifyTerminated(grpcHandler.connectionError());
        }
    });

    ChannelHandler negotiationHandler = protocolNegotiator.newHandler
    (grpcHandler);
    channel.pipeline().addLast(negotiationHandler);
}
```

(3) NettyServerTransport 的 start 方法创建了 NettyServerHandler 实例, NettyServerHandler 是 gRPC 用来处理 HTTP/2 的 ChannelHandler, 它的类继承关系如图 14-5 所示。



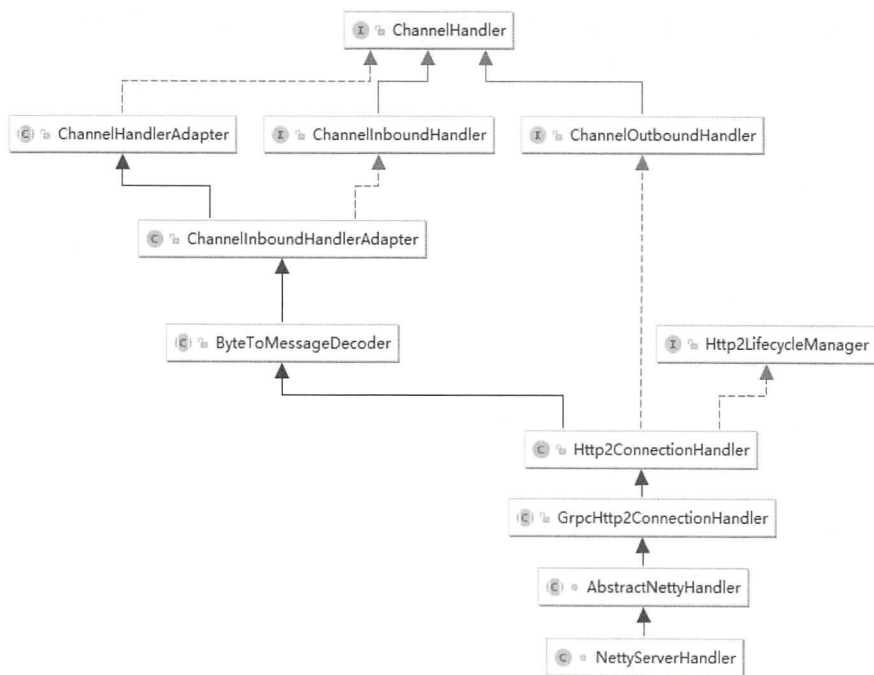


图 14-5 NettyServerHandler 的类继承关系

从类继承关系可以看出，NettyServerHandler 主要负责 HTTP/2 消息的处理，例如 HTTP/2 请求消息体和消息头的读取、Frame 消息的发送、Stream 状态消息的处理等，相关接口的定义如图 14-6 所示。

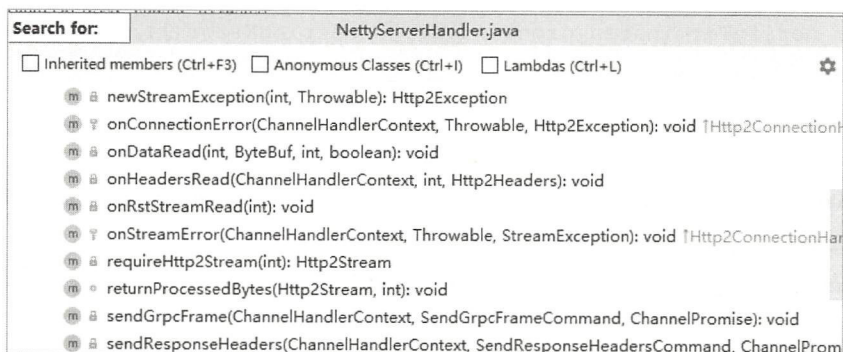


图 14-6 NettyServerHandler 处理 HTTP/2 消息相关接口的定义

(4) 创建 ProtocolNegotiator 实例，用于 HTTP/2 连接创建的协商。gRPC 支持三种协



商策略，分别是 PlaintextNegotiator、PlaintextUpgradeNegotiator 和 TlsNegotiator。其中 PlaintextUpgradeNegotiator 通过设置 Http2ClientUpgradeCodec，用于 101 协商和协议升级，相关代码实现如下（PlaintextUpgradeNegotiator 类）：

---

```
public Handler newHandler(GrpcHttp2ConnectionHandler handler) {  
    Http2ClientUpgradeCodec upgradeCodec = new Http2ClientUpgradeCodec  
(handler);  
    HttpClientCodec httpClientCodec = new HttpClientCodec();  
    final HttpClientUpgradeHandler upgrader =  
        new HttpClientUpgradeHandler(httpClientCodec, upgradeCodec, 1000);  
    return new BufferingHttp2UpgradeHandler(upgrader);  
}
```

---

（5）创建 ServerBootstrap、bossGroup 和 workerGroup，启动 HTTP/2 服务端，代码如下（NettyServer 类）：

---

```
public void start(ServerListener serverListener) throws IOException {  
    listener = checkNotNull(serverListener, "serverListener");  
    allocateSharedGroups();  
    ServerBootstrap b = new ServerBootstrap();  
    b.group(bossGroup, workerGroup);  
    b.channel(channelType);  
    if (NioServerSocketChannel.class.isAssignableFrom(channelType)) {  
        b.option(SO_BACKLOG, 128);  
        b.childOption(SO_KEEPALIVE, true);  
    }  
    b.childHandler(new ChannelInitializer<Channel>() {  
        @Override  
        public void initChannel(Channel ch) throws Exception {  
            long maxConnectionAgeInNanos = NettyServer.this.maxConnectionAgeInNanos;  
            if (maxConnectionAgeInNanos != MAX_CONNECTION_AGE_NANOS_DISABLED) {  
                maxConnectionAgeInNanos =  
                    (long) ((.9D + Math.random() * .2D) * maxConnectionAgeInNanos);  
            }  
        }  
    })  
}
```

---

```
//代码省略
transport.start(transportListener);
//代码省略
ChannelFuture future = b.bind(address);
//代码省略
}
```

Netty HTTP/2 服务端创建流程如图 14-7 所示。

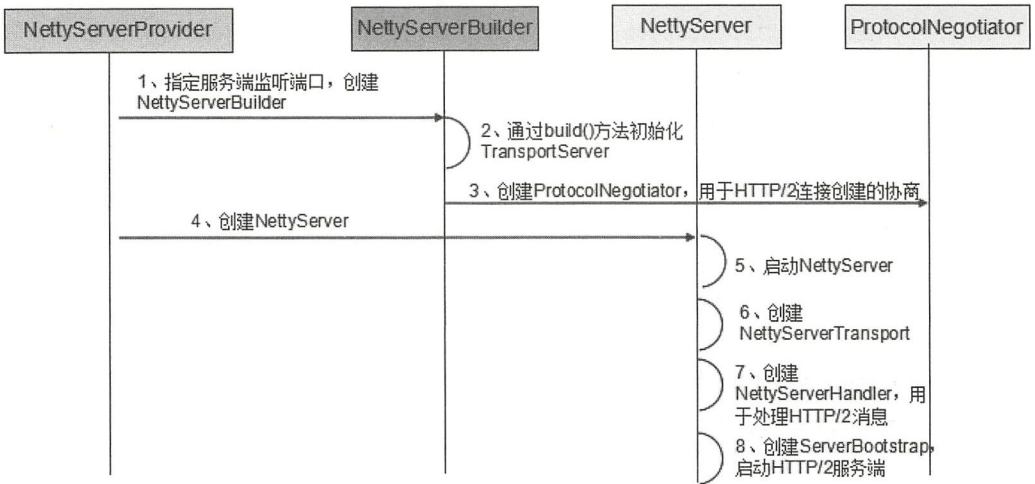


图 14-7 Netty HTTP/2 服务端创建流程

### 14.2.2 服务端接收 HTTP/2 请求消息原理和源码分析

gRPC 服务端的请求消息由 Netty HTTP/2 协议栈负责接入，gRPC 通过继承 `Http2FrameAdapter`，将自定义的 `FrameListener` 添加到 Netty 的 `Http2ConnectionDecoder` 中，在 HTTP/2 请求消息头和消息体被解析成功之后，回调 gRPC 的 `FrameListener`，接收并处理 HTTP/2 请求消息，如图 14-8 所示。

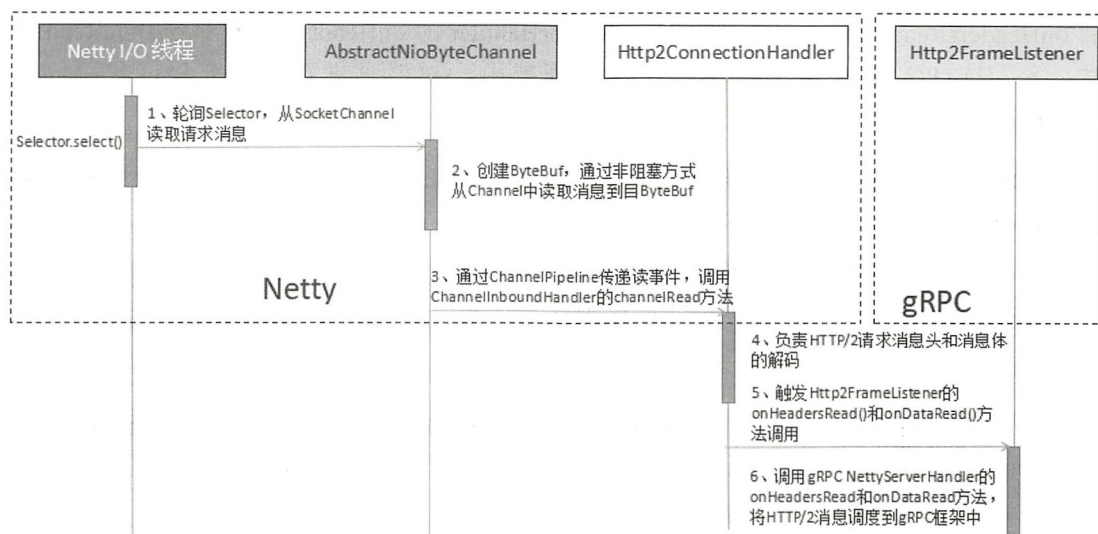


图 14-8 gRPC Netty HTTP/2 请求消息接入流程

gRPC 注册 FrameListener 相关源码如下（NettyServerHandler 类）：

```

private NettyServerHandler(
    ServerTransportListener transportListener,
    List<ServerStreamTracer.Factory> streamTracerFactories,
    Http2ConnectionDecoder decoder,
    Http2ConnectionEncoder encoder, Http2Settings settings,
    int maxMessageSize,
    long keepAliveTimeInNanos,
    long keepAliveTimeoutInNanos,
    long maxConnectionAgeInNanos,
    long maxConnectionAgeGraceInNanos,
    KeepAliveEnforcer keepAliveEnforcer) {
    super(decoder, encoder, settings);
    //代码省略
    decoder().frameListener(new FrameListener());
}
  
```

当 Netty HTTP/2 协议栈成功读取到请求消息时，会调用 FrameListener 的 onDataRead

和 `onHeadersRead` 方法，最后调用 `NettyServerHandler` 的 `onHeadersRead` 和 `onDataRead` 方法，将 HTTP/2 请求消息调度到 gRPC 框架中，对消息进行反序列化，代码如下（`FrameListener` 类）：

---

```
public int onDataRead(ChannelHandlerContext ctx, int streamId, ByteBuf data,
int padding,
    boolean endOfStream) throws Http2Exception {
    if (keepAliveManager != null) {
        keepAliveManager.onDataReceived();
    }
    NettyServerHandler.this.onDataRead(streamId, data, padding, endOfStream);
    return padding;
}

public void onHeadersRead(ChannelHandlerContext ctx,
    int streamId,
    Http2Headers headers,
    int streamDependency,
    short weight,
    boolean exclusive,
    int padding,
    boolean endStream) throws Http2Exception {
    if (keepAliveManager != null) {
        keepAliveManager.onDataReceived();
    }
    NettyServerHandler.this.onHeadersRead(ctx, streamId, headers);
}
```

---

## 1. HTTP/2 消息头的读取和处理

gRPC HTTP/2 消息头的读取和处理流程如图 14-9 所示。



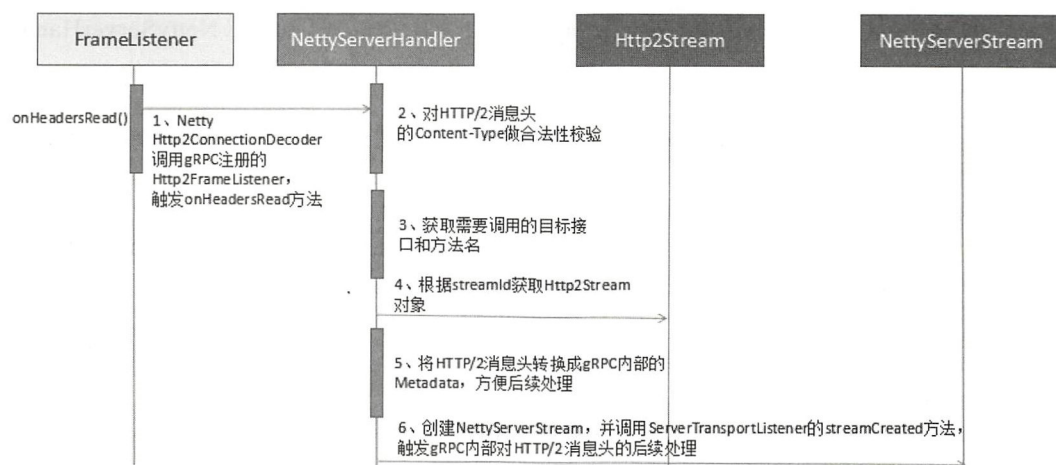


图 14-9 gRPC HTTP/2 消息头的读取和处理流程

主要的处理流程如下。

(1) Netty HTTP/2 协议栈完成消息头的解码之后, 触发 gRPC 注册的 FrameListener, 最终调用 NettyServerHandler 的 onHeadersRead 方法, 处理消息头。

(2) 对 HTTP/2 消息头的 Content-Type 字段做合法性校验, 如果不是“application/grpc”则非法, 抛出 Http2Exception 异常, 代码如下 (NettyServerHandler 类):

---

```

private void verifyContentType(int streamId, Http2Headers headers) throws
Http2Exception {
    CharSequence contentType = headers.get(CONTENT_TYPE_HEADER);
    if (contentType == null) {
        throw Http2Exception.streamError(streamId, Http2Error.REFUSED_STREAM,
            "Content-Type is missing from the request");
    }
    String contentTypeString = contentType.toString();
    if (!GrpcUtil.isGrpcContentType(contentTypeString)) {
        throw Http2Exception.streamError(streamId, Http2Error.REFUSED_STREAM,
            "Content-Type '%s' is not supported", contentTypeString);
    }
}

```

---

(3) 从 HTTP 消息头的 URL 中提取目标接口和方法名, 以 gRPC 官方 Demo HelloWorldServer

为例,它的 method 为“helloworld.Greeter/SayHello”,方法提取代码如下(NettyServerHandler 类):

---

```
private String determineMethod(int streamId, Http2Headers headers) throws
HttpException {
    if (!HTTP_METHOD.equals(headers.method())) {
        throw Http2Exception.streamError(streamId, Http2Error.REFUSED_STREAM,
            "Method '%s' is not supported", headers.method());
    }
    CharSequence path = headers.path();
    if (path.charAt(0) != '/') {
        throw Http2Exception.streamError(streamId, Http2Error.REFUSED_STREAM,
            "Malformatted path: %s", path);
    }
    return path.subSequence(1, path.length()).toString();
}
```

---

(4)根据 streamId 获取缓存的 Netty Http2Stream 对象,如果获取结果为空,说明 streamId 非法,抛出异常,代码如下 (NettyServerHandler 类):

---

```
private Http2Stream requireHttp2Stream(int streamId) {
    Http2Stream stream = connection().stream(streamId);
    if (stream == null) {
        throw new AssertionError("Stream does not exist: " + streamId);
    }
    return stream;
}
```

---

(5) 将 Netty 的 HTTP/2 消息头转换成 gRPC 内部的 Metadata, Metadata 内部维护了一个键值对的二维数组 namesAndValues, 以及一系列的类型转换方法, 相关代码如下 (NettyServerHandler 类):

---

```
public static Metadata convertHeaders(Http2Headers http2Headers) {
    if (http2Headers instanceof GrpcHttp2InboundHeaders) {
        GrpcHttp2InboundHeaders h = (GrpcHttp2InboundHeaders) http2Headers;
        return InternalMetadata.newMetadata(h.numHeaders(), h.namesAndValues());
    }
}
```

---

```

    return InternalMetadata.newMetadata(convertHeadersToArray(http2Headers));
}

```

(6) 创建 `NettyServerStream`，它聚合 `Sink`、`TransportState` 等类，通过 `transportListener` 的 `streamCreated` 方法对消息头进行处理，相关代码如下（`NettyServerHandler` 类）：

```

private void onHeadersRead(ChannelHandlerContext ctx, int streamId,
    Http2Headers headers) throws Http2Exception {
    //代码省略

    NettyServerStream.TransportState state = new NettyServerStream.TransportState(
        this, http2Stream, maxMessageSize, statsTraceCtx);
    String authority = getOrUpdateAuthority((AsciiString) headers.authority());
    NettyServerStream stream = new NettyServerStream(ctx.channel(), state,
        attributes, authority, statsTraceCtx);
    transportListener.streamCreated(stream, method, metadata);
    //代码省略
}

```

## 2. HTTP/2 消息体的处理

消息体的处理入口仍然是 gRPC 注册的 `FrameListener`，通过调用 `NettyServerHandler` 的 `onDataRead` 方法完成消息体的读取，gRPC HTTP/2 消息体处理流程如图 14-10 所示。

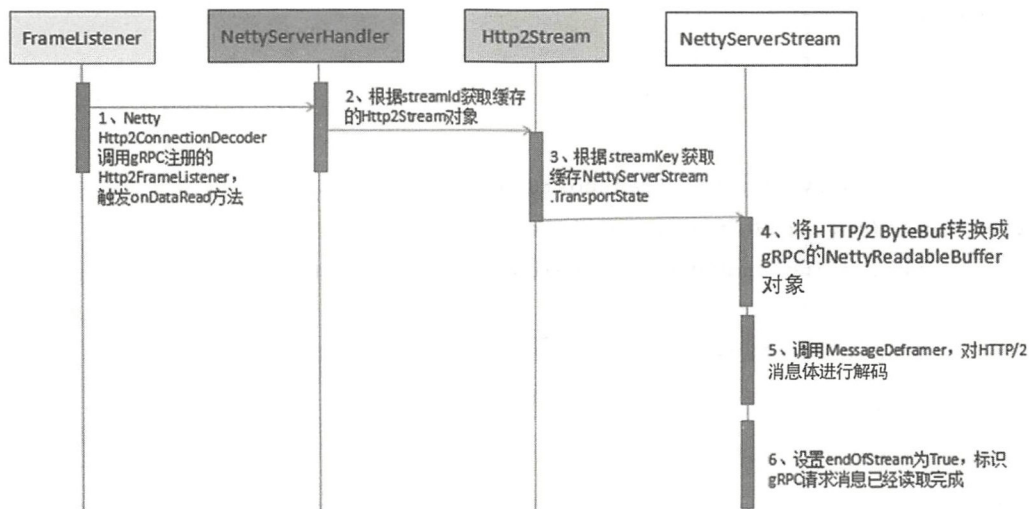


图 14-10 gRPC HTTP/2 消息体处理流程

主要的处理流程介绍如下。

(1) 对 HTTP/2 的 flow control pinging 和 flow control window updates 消息进行处理，代码如下（FlowControlPinger 类）：

---

```
public void onDataRead(int dataLength, int paddingLength) {
    if (!autoTuneFlowControlOn) {
        return;
    }
    if (!isPinging()) {
        setPinging(true);
        sendPing(ctx());
    }
    incrementDataSincePing(dataLength + paddingLength);
}
```

---

(2) 根据 streamId 获取 Http2Stream，然后根据 Http2Stream 的 Http2Connection.PropertyKey，获取 NettyServerStream 的 TransportState 类，触发 gRPC 的消息读取流程，相关代码如下（NettyServerHandler 类）：

---

```
private void onDataRead(int streamId, ByteBuf data, int padding, boolean
endOfStream)
    throws Http2Exception {
    flowControlPing().onDataRead(data.readableBytes(), padding);
    try {
        NettyServerStream.TransportState stream =
serverStream(requireHttp2Stream(streamId));
        stream.inboundDataReceived(data, endOfStream);
    } catch (Throwable e) {
        logger.log(Level.WARNING, "Exception in onDataRead()", e);
        throw new StreamException(streamId, e);
    }
}
```

---

(3) 将 Netty 的请求消息体 ByteBuf 转换成 gRPC 内部的 NettyReadableBuffer 对象，

代码如下 (TransportState 类):

---

```
void inboundDataReceived(ByteBuf frame, boolean endOfStream) {
    super.inboundDataReceived(new NettyReadableBuffer(frame.retain()),
endOfStream);
}
```

---

(4)调用 deframe 方法,完成请求消息体的解码,相关代码如下 (AbstractStream2 类):

---

```
protected final void deframe(ReadableBuffer frame, boolean endOfStream) {
    if (deframer.isClosed()) {
        frame.close();
        return;
    }
    try {
        deframer.deframe(frame, endOfStream);
    } catch (Throwable t) {
        deframeFailed(t);
    }
}
```

---

### 14.2.3 服务端发送 HTTP/2 响应消息原理和源码分析

gRPC 服务端通过将响应消息封装成 WriteQueue.AbstractQueuedCommand, 异步写入 WriteQueue, 然后调用 WriteQueue 的 scheduleFlush 操作, 将响应消息发送命令放到 NioEventLoop 中执行, 调用 channel.write 方法将其发送到 ChannelPipeline, 由 gRPC 的 NettyServerHandler 拦截 write 方法, 按照命令的分类进行处理, 最后调用 Netty Http2ConnectionEncoder 的 writeXXX 方法完成响应消息的发送。gRPC 服务端响应消息发送流程如图 14-11 所示。



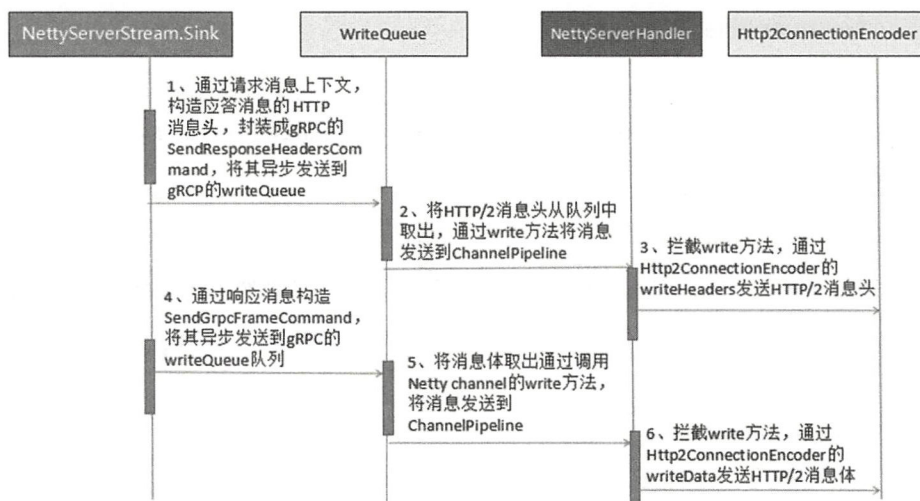


图 14-11 gRPC 服务端响应消息发送流程

关键流程解读如下。

(1) 通过 `NettyServerStream` 的 `Sink` 类对需要发送的 HTTP/2 响应消息进行 Task 封装，实现消息发送的异步化，代码如下（`NettyServerStream.Sink` 类）：

```

public void writeHeaders(Metadata headers) {
    writeQueue.enqueue(new SendResponseHeadersCommand(transportState(),
        Utils.convertServerHeaders(headers), false),
        true);
}

public void writeFrame(WritableBuffer frame, boolean flush) {
    if (frame == null) {
        writeQueue.scheduleFlush();
        return;
    }
    ByteBuf bytebuf = ((NettyWritableBuffer) frame).bytebuf();
    final int numBytes = bytebuf.readableBytes();
    onSendingBytes(numBytes);
    writeQueue.enqueue(
        new SendGrpcFrameCommand(transportState(), bytebuf, false),

```

```

        channel.newPromise().addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                transportState().onSentBytes(numBytes);
            }
        }), flush);
    }
}

```

---

(2) WriteQueue 将任务投递到对应 Channel 的 NioEventLoop 线程异步执行消息发送操作，代码如下 (WriteQueue 类)：

```

void scheduleFlush() {
    if (scheduled.compareAndSet(false, true)) {
        channel.eventLoop().execute(later);
    }
}

```

---

(3) NioEventLoop 线程循环处理待发送的响应消息，调用 Channel 的 write 方法，将消息发送到 ChannelPipeline，由 gRPC ChannelHandler 拦截处理 (WriteQueue 类)：

```

private void flush() {
    try {
        QueuedCommand cmd;
        int i = 0;
        boolean flushedOnce = false;
        while ((cmd = queue.poll()) != null) {
            channel.write(cmd, cmd.promise());
            if (++i == DEQUE_CHUNK_SIZE) {
                i = 0;
                channel.flush();
                flushedOnce = true;
            }
        }
        //后续代码省略
    }
}

```

---

(4) gRPC 的 `NettyServerHandler` 拦截 `write` 方法，按照命令的类型进行分类处理，如果是 `SendGrpcFrameCommand` 和 `SendResponseHeadersCommand`，则调用 Netty 的 `Http2ConnectionEncoder` 完成 HTTP/2 消息的发送，相关代码如下(`NettyServerHandler`类)：

---

```

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise)
    throws Exception {
    if (msg instanceof SendGrpcFrameCommand) {
        sendGrpcFrame(ctx, (SendGrpcFrameCommand) msg, promise);
    } else if (msg instanceof SendResponseHeadersCommand) {
        sendResponseHeaders(ctx, (SendResponseHeadersCommand) msg, promise);
    } else if (msg instanceof CancelServerStreamCommand) {
        cancelStream(ctx, (CancelServerStreamCommand) msg, promise);
    } else if (msg instanceof ForcefulCloseCommand) {
        forcefulClose(ctx, (ForcefulCloseCommand) msg, promise);
    } else {
        AssertionError e =
            new AssertionError("Write called for unexpected type: " +
msg.getClass().getName());
        ReferenceCountUtil.release(msg);
        promise.setFailure(e);
        throw e;
    }
}

```

---

## 14.3 gRPC Netty HTTP/2 客户端工作机制

gRPC 的客户端调用主要包括基于 Netty 的 HTTP/2 客户端创建、客户端负载均衡、请求消息的发送和响应接收处理 4 个流程。由于客户端负载均衡与 Netty 无关，因此本章不做介绍。

### 14.3.1 Netty HTTP/2 客户端创建原理和源码分析

gRPC 客户端底层基于 Netty 4.1.X 的 HTTP/2 协议栈构建，通过 HTTP/2 承载 RPC 请求和响应消息，能够解决传统 HTTP/1.1 存在的性能问题，相比 RPC 私有协议，它的规范性更好一些。

gRPC HTTP/2 协议栈（客户端）的主要实现是 NettyClientTransport 和 NettyClientHandler，gRPC 的 Netty HTTP/2 客户端创建流程如图 14-12 所示。

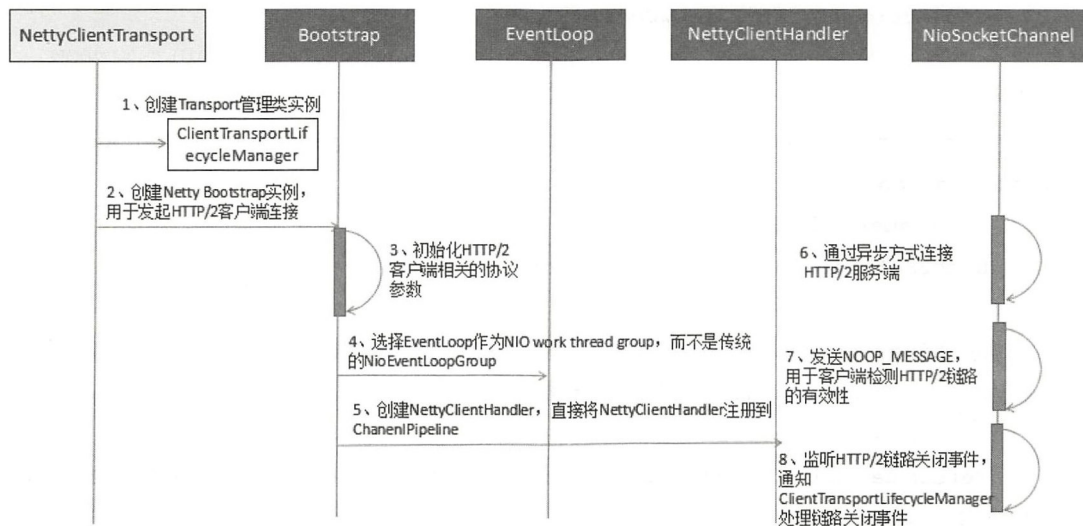


图 14-12 gRPC 的 Netty HTTP/2 客户端创建流程

关键流程和源码解读如下。

(1) 通过 NettyClientTransport 的 start 方法创建 NettyClientHandler，NettyClientHandler 初始化代码如下（NettyClientHandler 类）：

```

static NettyClientHandler newHandler(Http2Connection connection,
Http2FrameReader frameReader,
    Http2FrameWriter frameWriter, ClientTransportLifecycleManager
lifecycleManager,
    KeepAliveManager keepAliveManager, int flowControlWindow, int
maxHeaderListSize,

```

```

        Ticker ticker, Runnable tooManyPingsRunnable) {
//代码省略
        Http2FrameLogger frameLogger = new Http2FrameLogger(LogLevel.DEBUG,
NettyClientHandler.class);
        frameReader = new Http2InboundFrameLogger(frameReader, frameLogger);
        frameWriter = new Http2OutboundFrameLogger(frameWriter, frameLogger);
        StreamBufferingEncoder encoder = new StreamBufferingEncoder(
            new DefaultHttp2ConnectionEncoder(connection, frameWriter));
        connection.local().flowController(
            new DefaultHttp2LocalFlowController(connection, DEFAULT_WINDOW_
UPDATE_RATIO, true));
        Http2ConnectionDecoder decoder = new FixedHttp2ConnectionDecoder
(connection, encoder,
            frameReader);
        Http2Settings settings = new Http2Settings();
        settings.pushEnabled(false);
        settings.initialWindowSize(flowControlWindow);
        settings.maxConcurrentStreams(0);
        settings.maxHeaderListSize(maxHeaderListSize);
        return new NettyClientHandler(decoder, encoder, settings, lifecycleManager,
keepAliveManager,
            ticker, tooManyPingsRunnable);
    }

```

(2) 通过 NettyClientHandler 创建 HTTP/2 协商类 negotiationHandler，如图 14-13 所示。

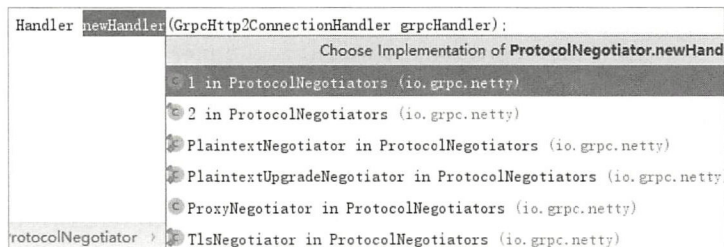


图 14-13 创建 HTTP/2 协商类



(3) 创建客户端 Bootstrap, 发起 HTTP/2 连接, 代码如下 (NettyClientTransport 类):

---

```

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise)
    throws Exception {
    Bootstrap b = new Bootstrap();
    b.group(eventLoop);
    b.channel(channelType);
    if (NioSocketChannel.class.isAssignableFrom(channelType)) {
        b.option(SO_KEEPALIVE, true);
    }
    //代码省略
    b.handler(negotiationHandler);
    ChannelFuture regFuture = b.register();
    channel = regFuture.channel();
    //代码省略
    channel.connect(address).addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws
Exception {
            if (!future.isSuccess()) {
                ChannelHandlerContext ctx = future.channel().pipeline().
context(handler);
                if (ctx != null) {
                    ctx.fireExceptionCaught(future.cause());
                }

                future.channel().pipeline().fireExceptionCaught(future.cause());
            }
        }
    });
    //代码省略
}

```

---

gRPC Netty HTTP/2 客户端的创建与传统客户端的创建存在一些差异，主要体现在如下两点。

- ◎ 创建 `NettyClientHandler`（实际被包装成 `ProtocolNegotiator.Handler`，用于 HTTP/2 的握手协商）之后，不是由传统的 `ChannelInitializer` 在初始化 `Channel` 时将 `NettyClientHandler` 加入 `Pipeline`，而是直接通过 `Bootstrap` 的 `Handler` 方法加入 `Pipeline`，以便立即接收和发送任务。
- ◎ 客户端使用的 `work` 线程组并非通常意义的 `EventLoopGroup`，而是一个 `EventLoop`，即 HTTP/2 客户端使用的 `work` 线程并非一组线程（默认线程数为“CPU 内核数  $\times 2$ ”），而是一个 `EventLoop` 线程。这其实也很容易理解，一个 `NioEventLoop` 线程可以同时处理多个 HTTP/2 客户端连接，它是多路复用的，对于单个 HTTP/2 客户端，如果默认独占一个 `work` 线程组，将造成极大的资源浪费，同时也可能导致句柄溢出（并发启动大量 HTTP/2 客户端）。

（4）Netty 的 `NioSocketChannel` 初始化并向 `Selector` 注册之后（发起 HTTP 连接之前），立即由 `NettyClientHandler` 创建 `WriteQueue`，用于接收并处理 gRPC 内部的各种指令，例如链路关闭指令、发送 `Frame` 指令、发送 `Ping` 指令等。相关代码如下（`NettyClientHandler` 类）：

---

```
void startWriteQueue(Channel channel) {  
    clientWriteQueue = new WriteQueue(channel);  
}
```

---

### 14.3.2 客户端发送 HTTP/2 请求消息原理和源码分析

gRPC 默认基于“Netty HTTP/2 + Protocol buffer”进行 RPC 调用，gRPC HTTP/2 客户端请求消息发送流程如图 14-14 所示。

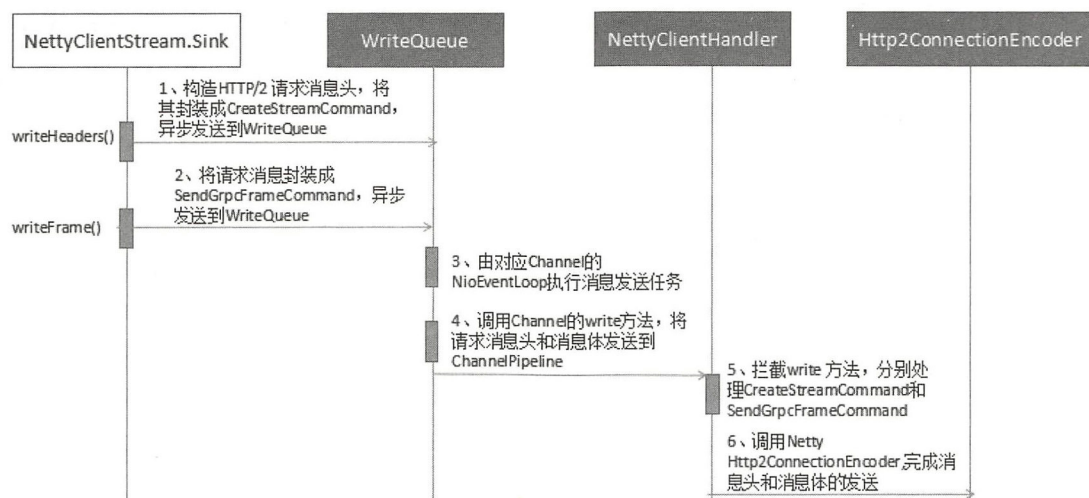


图 14-14 gRPC HTTP/2 客户端请求消息发送流程

关键技术点说明如下。

(1) 在客户端发送 HTTP/2 请求消息头时，将 gRPC 内部 Metadata 格式的消息头转换成 Netty 的 Http2Headers，转换后将请求消息头封装成 CreateStreamCommand，发送到 WriteQueue，代码如下（NettyClientStream.Sink 类）：

```

public void writeHeaders(Metadata headers, byte[] requestPayload) {
    //代码省略

    Http2Headers http2Headers = Utils.convertClientHeaders(headers,
scheme, defaultPath, authority, httpMethod, userAgent);

    //代码省略

    writeQueue.enqueue(new CreateStreamCommand(http2Headers, transportState(),
get), !method.getType().clientSendsOneMessage() || get).addListener
(failureListener);
}
  
```

(2) 将请求消息封装成 SendGrpcFrameCommand 发送到 WriteQueue，代码如下（NettyClientStream.Sink 类）：

```

public void writeFrame(WritableBuffer frame, boolean endOfStream, boolean
flush) {
  
```

```

        ByteBuf bytebuf = frame == null ? EMPTY_BUFFER : ((NettyWritableBuffer)
frame).bytebuf();
        final int numBytes = bytebuf.readableBytes();
        if (numBytes > 0) {
            onSendingBytes(numBytes);
            writeQueue.enqueue(
                new SendGrpcFrameCommand(transportState(), bytebuf, endOfStream),
                channel.newPromise().addListener(new ChannelFutureListener() {
                    @Override
                    public void operationComplete(ChannelFuture future) throws
Exception {
                        if (future.isSuccess()) {
                            transportState().onSentBytes(numBytes);
                        }
                    }
                })), flush);
        } else {
            writeQueue.enqueue(new SendGrpcFrameCommand(transportState(), bytebuf,
endOfStream), flush);
        }
    }
}

```

---

(3) NettyClientHandler 拦截 write 方法，分别处理 CreateStreamCommand 和 SendGrpcFrameCommand，代码如下（NettyClientHandler 类）：

---

```

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise)
    throws Exception {
    if (msg instanceof CreateStreamCommand) {
        createStream((CreateStreamCommand) msg, promise);
    } else if (msg instanceof SendGrpcFrameCommand) {
        sendGrpcFrame(ctx, (SendGrpcFrameCommand) msg, promise);
    } else if (msg instanceof CancelClientStreamCommand) {
        cancelStream(ctx, (CancelClientStreamCommand) msg, promise);
    }
}

```

```

    } else if (msg instanceof SendPingCommand) {
        sendPingFrame(ctx, (SendPingCommand) msg, promise);
    } else if (msg instanceof GracefulCloseCommand) {
        gracefulClose(ctx, (GracefulCloseCommand) msg, promise);
    } else if (msg instanceof ForcefulCloseCommand) {
        forcefulClose(ctx, (ForcefulCloseCommand) msg, promise);
    } else if (msg == NOOP_MESSAGE) {
        ctx.write(Unpooled.EMPTY_BUFFER, promise);
    } else {
        throw new AssertionError("Write called for unexpected type: " +
            msg.getClass().getName());
    }
}

```

---

(4) 发送客户端请求消息头时，需要创建 `NettyClientStream`，通过自增的方式生成 `streamId`，然后创建 `NettyClientStream` 的 `TransportState` 对象，相关代码如下(`NettyClientHandler` 类)：

```

private void createStream(CreateStreamCommand command, final ChannelPromise
promise)

    throws Exception {
    if (lifecycleManager.getShutdownThrowable() != null) {
        promise.setFailure(lifecycleManager.getShutdownThrowable());
        return;
    }
    final int streamId;
    try {
        streamId = incrementAndGetNextStreamId();
    } catch (StatusException e) {
        //代码省略
        return;
    }
    final NettyClientStream.TransportState stream = command.stream();
    final Http2Headers headers = command.headers();

```





```

stream.setId(streamId);
//代码省略
}

```

(5) 创建 stream 后，调用 Netty Http2ConnectionEncoder 的 writeHeaders 和 writeData 方法，完成请求消息头和消息体的发送。

### 14.3.3 客户端接收 HTTP/2 响应消息原理和源码分析

gRPC 客户端响应消息的接收入口是 NettyClientHandler，gRPC HTTP/2 客户端读取响应消息流程如图 14-15 所示。

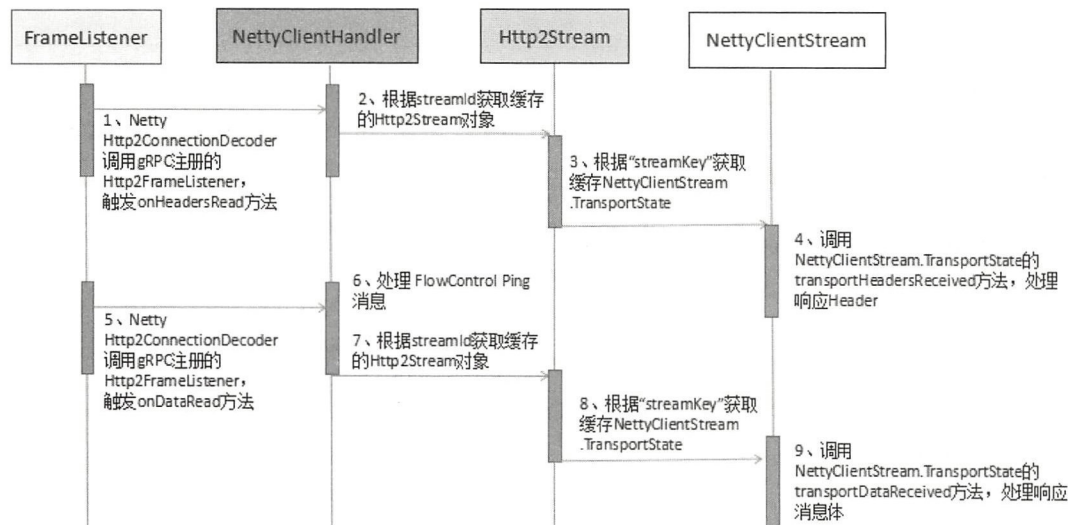


图 14-15 gRPC HTTP/2 客户端读取响应消息流程

关键技术点说明如下。

(1) NettyClientHandler 的 onHeadersRead(int streamId, Http2Headers headers, boolean endStream)方法会被调用两次，根据 endStream 判断是否是 Stream 的结尾。

(2) 请求和响应的映射关系：根据 streamId 可以关联同一个 HTTP/2 Stream，将 NettyClientStream 缓存到 Stream，客户端就可以在接收到响应消息头或消息体时还原



NettyClientStream，进行后续处理。

(3) 客户端和服务端的 HTTP/2 Header 和 Data Frame 解析共用同一个方法，即 MessageDeframer 的 deliver 方法。

## 14.4 gRPC 消息序列化机制

gRPC 默认支持 Protocol buffer 和 JSON 两种序列化机制，它的消息序列化和反序列化并没有使用 Netty 提供的 CodeC 框架，而是由 gRPC 业务线程接收 Netty 传递的 HTTP/2 消息后自己做序列化和反序列化。

### 14.4.1 Google Protobuf 简介

Google 的 Protobuf 在业界应用得比较广泛，很多项目选择 Protobuf 作为序列化框架，当前最新的版本为 Protocol buffer 3，具有如下几个优点。

- (1) Google 内部长期使用，产品成熟度和可靠性高。
- (2) 支持多种语言，包括 C++、Java、Python 和 PHP 等。
- (3) 编码后的消息更小，有利于存储和网络传输。
- (4) 序列化和反序列化的性能非常高。
- (5) 支持不同消息版本的前向兼容。
- (6) 支持自定义可选和必选字段。

Protobuf 是一个灵活、高效、结构化的数据序列化框架，相比 XML 等传统的序列化工具，它更小、更快、更简单。Protobuf 支持跨语言使用，定义 proto 文件后，可以通过代码生成工具自动生成不同语言版本的源代码，甚至可以在使用不同版本的数据结构进程间进行数据传递，实现数据结构的前向兼容。



## 14.4.2 消息的序列化原理和源码分析

gRPC 在将消息发送到 `WriteQueue` 之前，会调用 `requestMarshaller` 将消息序列化成 `InputStream`，以客户端发送请求消息为例进行说明，调用 `ClientCallImpl` 的 `sendMessage`，发送请求消息，实际上并未真正发送消息，而是使用 `Protocol buffer` 对消息做序列化，代码如下（`ClientCallImpl` 类）：

---

```
public void sendMessage(ReqT message) {
    Preconditions.checkNotNull(stream != null, "Not started");
    Preconditions.checkNotNull(!cancelCalled, "call was cancelled");
    Preconditions.checkNotNull(!halfCloseCalled, "call was half-closed");
    try {
        InputStream messageIs = method.streamRequest(message);
        stream.writeMessage(messageIs);
        //后续代码省略
    }
}
```

---

序列化的具体实现如下（`ProtoLiteUtils` 类）：

---

```
public static <T extends MessageLite> Marshaller<T> marshaller(final T
defaultInstance) {
    final Parser<T> parser = defaultInstance.getParserForType();
    return new PrototypeMarshaller<T>() {
        public Class<T> getMessageClass() {
            return defaultInstance.getClass();
        }
        public T getMessagePrototype() {
            return defaultInstance;
        }
        public InputStream stream(T value) {
            return new ProtoInputStream(value, parser);
        }
        //后续代码省略
    }
}
```

---



完成序列化后，调用 stream 的 writeMessage 方法，继续进行消息的发送。

### 14.4.3 消息的反序列化原理和源码分析

Netty 通过 gRPC 注册的 FrameListener 回调 NettyClientHandler 和 NettyServerHandler，对消息进行处理。以服务端请求消息反序列化为例进行说明，由 gRPC 的 SerializingExecutor 负责消息体的解析，代码如下（JumpToApplicationThreadServerStreamListener 类）：

---

```
public void messageRead(final InputStream message) {
    callExecutor.execute(new ContextRunnable(context) {
        @Override
        public void runInContext() {
            try {
                getListener().messageRead(message);
            } catch (RuntimeException e) {
                internalClose(Status.fromThrowable(e), new Metadata());
                throw e;
            } catch (Error e) {
                internalClose(Status.fromThrowable(e), new Metadata());
                throw e;
            }
        }
    });
}
```

---

最终调用 ProtoLiteUtils 的 Marshaller 方法，通过 parse(InputStream stream)方法将 NettyReadableBuffer 反序列化为原始的请求对象，代码如下（ProtoLiteUtils 类）：

---

```
public T parse(InputStream stream) {
    if (stream instanceof ProtoInputStream) {
        ProtoInputStream protoStream = (ProtoInputStream) stream;
        if (protoStream.parser() == parser) {
            try {
```

---



```
        T message = (T) ((ProtoInputStream) stream).message();
        return message;
    } catch (IllegalStateException ex) {
    }
}

//代码省略
}
```

---

## 14.5 gRPC 线程模型

影响 RPC 框架性能的三个核心要素如下。

(1) I/O：用什么样的 I/O 方式将数据发送给对方，BIO、NIO 或者 AIO，I/O 模型在很大程度上决定了框架通信的性能。

(2) 协议：采用什么样的通信协议，公有协议还是私有协议，采用 JSON 序列化还是其他二进制序列化框架，决定了消息的传输效率。

(3) 线程模型：消息读取之后的编解码在哪个线程中进行，编解码后的消息如何派发，通信线程模型不同，对性能的影响也不同。

在以上三个要素中，线程模型对性能的影响非常大。随着硬件性能的提升，CPU 的核数越来越多，很多服务器标配已经达到 64 核，通过多线程并发编程，可以充分利用多核 CPU 的处理能力，提升系统的处理效率和并发性能。但是如果线程创建或者管理不当，频繁发生线程上下文切换或者锁竞争，反而会影响系统的性能。

### 14.5.1 服务端线程模型

gRPC 服务端线程模型如图 14-16 所示。





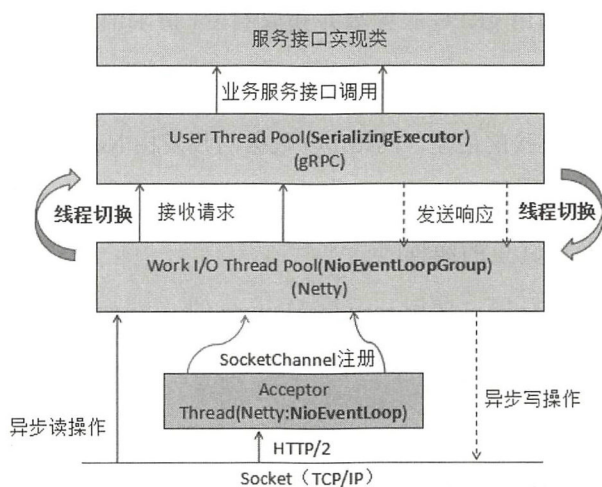


图 14-16 gRPC 服务端线程模型

其中，HTTP/2 服务端创建、HTTP/2 请求消息的接入和响应发送都由 Netty NioEventLoop 线程负责，gRPC 消息的序列化和反序列化业务服务接口的调用由 gRPC 的 SerializingExecutor 线程池负责。

## 14.5.2 客户端线程模型

以同步阻塞调用方式为例，gRPC 客户端线程模型如图 14-17 所示。

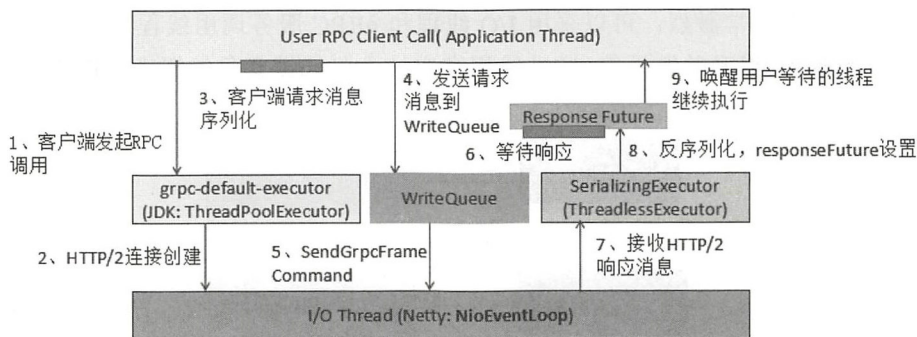


图 14-17 gRPC 客户端线程模型（同步阻塞调用方式）

客户端调用主要涉及的线程如下。



- (1) 业务线程，负责调用 gRPC 服务端并获取响应，请求消息的序列化由该线程负责。
- (2) 客户端负载均衡策略及 Netty 客户端的创建由 `grpc-default-executor` 线程池负责。
- (3) HTTP/2 客户端连接的创建、网络 I/O 数据的收发由 Netty `NioEventLoop` 线程负责。
- (4) HTTP/2 响应消息的反序列化由 `SerializingExecutor` 负责，它的具体实现是 `ThreadlessExecutor`，并非真正的线程池。
- (5) `SerializingExecutor` 通过调用 `responseFuture` 的 `set(result)`，唤醒阻塞的业务线程继续执行，完成同步阻塞方式的 RPC 调用。

### 14.5.3 线程模型总结

消息的序列化和反序列化均由 gRPC 线程负责，而没有在 Netty 的 Handler 中做 `CodeC`，这样可以降低 Netty I/O 线程的工作负载，提升系统的吞吐量。

gRPC 采用的是网络 I/O 线程和业务调用线程分离的策略，在大部分场景下该策略是最优的。但是，对于那些接口逻辑非常简单，执行时间很短，不需要与外部网元交互、访问数据库或本地存储，也不需要等待其他同步操作的场景，则建议直接在 Netty I/O 线程中调用接口，不需要再发送到后端的业务线程池，避免了线程上下文切换，同时也消除了线程并发问题。

当前 Netty NIO 线程和 gRPC 的 `SerializingExecutor` 之间没有映射关系，当线程数量比较多时，锁竞争会非常激烈，可以采用 I/O 线程和 gRPC 服务调用线程绑定的方式，降低出现锁竞争的概率，提升并发性能，通过线程绑定技术降低锁竞争的概率如图 14-18 所示。

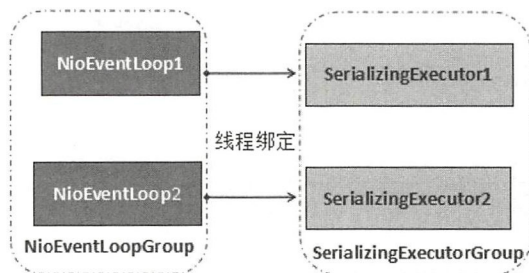


图 14-18 通过线程绑定技术降低锁竞争的概率



## 14.6 总结

Netty 4.1 提供了完整的 HTTP/2 协议栈，用户可以基于 Netty 4.1 框架快速开发支持 HTTP/2 的服务端应用。通过学习 Netty HTTP/2 协议栈在 gRPC 中的应用，可以掌握 HTTP/2 客户端和服务端的创建、HTTP/2 消息的收发及序列化和反序列化等功能，学习和借鉴 gRPC HTTP/2 处理的优点，可以在实际项目中少走很多弯路。



## 第 15 章

---

# Netty 事件触发策略使用不当案例

针对 Channel 上发生的各种网络操作，例如链路创建、链路关闭、消息读写、链路注册和去注册等，Netty 将这些消息封装成事件，触发 ChannelPipeline 调用 ChannelHandler 链，由系统或者用户实现的 ChannelHandler 对网络事件做处理。

由于网络事件种类比较多，触发和执行机制也存在一些差异，如果掌握不到位，很有可能遇到一些莫名其妙的问题。而且有些问题只有在高并发时或者在生产环境中才会出现，在测试环境中不容易复现，因此这类问题定位难度很大。

### 15.1 channelReadComplete 方法被调用多次问题

---

业务基于 Netty 开发了 HTTP 服务器，在生产环境中运行一段时间之后，部分消息逻辑处理错误，但是在灰度测试环境中却无法重现问题，需要尽快定位并解决问题。

#### 15.1.1 ChannelHandler 调用问题

---

在生产环境中将某一个服务实例的调测日志打开一段时间，以便定位问题。通过接口





日志分析发现，对于同一个 HTTP 请求消息，当发生问题时，业务 `ChannelHandler` 的 `channelReadComplete` 方法会被调用多次，但是大部分消息都调用一次，按照业务的设计初衷，当服务端读到一个完整的 HTTP 请求消息时，在 `channelReadComplete` 方法中进行业务逻辑处理。如果一个请求消息的 `channelReadComplete` 方法被调用多次，则业务逻辑会出现异常。

通过对客户端请求消息和 Netty 框架进行源码分析，找到了问题的根本原因：TCP 底层并不了解上层业务数据的具体含义，它会根据 TCP 缓冲区的实际情况进行包的拆分，所以在业务上认为一个完整的 HTTP 报文可能会被 TCP 拆分成多个包发送，也有可能把多个小的包封装成一个大的数据包发送。导致数据包拆分和重组的原因如下。

- (1) 应用程序写入的字节大小大于套接口发送缓冲区大小。
- (2) 进行 MSS 大小的 TCP 分段。
- (3) 以太网帧的有效载荷 (payload) 大于 MTU 的 IP 分片。
- (4) 开启了 TCP Nagle 算法。

由于底层的 TCP 无法理解上层的业务数据，所以在底层无法保证数据包不被拆分和重组，这个问题只能通过上层的应用协议栈设计来解决，根据业界主流协议的解决方案，归纳如下。

- (1) 消息定长，例如每个报文的大小固定为 200 字节，如果不够，空位补空格。
- (2) 在包尾增加换行符 (或者其他分隔符) 进行分隔，例如 FTP。
- (3) 将消息分为消息头和消息体，消息头包含表示消息总长度 (或者消息体长度) 的字段，通常消息头的第一个字段使用 `int32` 表示消息的总长度。

对于 HTTP 请求消息，当业务并发量比较大时，无法保证一个完整的 HTTP 消息会被一次全部读取到服务端。当采用 `chunked` 方式进行编码时，HTTP 报文也是分段发送的，此时服务端读取的也不是完整的 HTTP 报文。为了解决这个问题，Netty 提供了 `HttpObjectAggregator`，保证后端业务 `ChannelHandler` 接收的是一个完整的 HTTP 报文，相关示例代码如下：

---

```
//代码省略
ChannelPipeline p = ...;
```





```
p.addLast("decoder", new HttpRequestDecoder());
p.addLast("encoder", new HttpResponseEncoder());
p.addLast("aggregator", new HttpObjectAggregator(10240));
p.addLast("service", new ServiceChannelHandler());
//代码省略
```

`HttpObjectAggregator` 可以保证 Netty 读到完整的 HTTP 请求报文后才调用一次业务 `ChannelHandler` 的 `channelRead` 方法，无论这条报文底层经过了几次 `SocketChannel` 的 `read` 调用。但是 `channelReadComplete` 方法并不是在业务语义上的读取消息完成后被触发的，而是在每次从 `SocketChannel` 成功读到消息后，由系统触发，也就是说如果一个 HTTP 消息被 TCP 协议栈发送了  $N$  次，则服务端的 `channelReadComplete` 方法就会被调用  $N$  次。

在灰度测试环境中，由于客户端并没有采用 `chunked` 的编码方式，并发压力也不是很大，所以一直没有发现该问题，到了生产环境中有些客户端采用了 `chunked` 方式发送 HTTP 请求消息，客户端并发量也比较大，所以触发了服务端的问题。

### 15.1.2 生产环境问题模拟重现

对业务故障做模拟，思路为：采用分隔符“\$ \_”来区分消息，客户端每秒向服务端发送一个不完整的请求消息，到了 10s 的倍数时间时，发送分隔符“\$ \_”，模拟每 10s 发送一个完整的业务请求消息；对于服务端，分别在 `channelRead` 和 `channelReadComplete` 方法中打印日志，查看调用情况。

客户端代码如下（`EventTriggerClientHandler` 类）：

```
public class EventTriggerClientHandler extends ChannelInboundHandlerAdapter {
    private static AtomicInteger SEQ = new AtomicInteger(0);
    static final String ECHO_REQ = "Hi,welcome to Netty ";
    static final String DELIMITER = "$ _";
    static ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        scheduledExecutorService.scheduleAtFixedRate(()
```



```
->{
    int counter = SEQ.incrementAndGet();
    if (counter % 10 == 0)
    {
        ctx.writeAndFlush(Unpooled.copiedBuffer((ECHO_REQ +
DELIMITER).getBytes()));
    }
    else

ctx.writeAndFlush(Unpooled.copiedBuffer(ECHO_REQ.getBytes()));
    },0,1000, TimeUnit.MILLISECONDS);
}
//代码省略
```

---

服务端业务 ChannelHandler 的代码如下（EventTriggerServerHandler 类）：

---

```
public class EventTriggerServerHandler extends ChannelInboundHandlerAdapter {
    int counter;
    int readCompleteTimes;
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        String body = (String) msg;
        System.out.println("This is " + ++counter + " times receive client : ["
            + body + "]);
        body += "$_";
        ByteBuf echo = Unpooled.copiedBuffer(body.getBytes());
        ctx.writeAndFlush(echo);
    }
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
        ctx.fireChannelReadComplete();
        readCompleteTimes++;
    }
}
```



```
        System.out.println("This is " + readCompleteTimes + " times receive  
ReadComplete event.");  
    }  
    //代码省略
```

---

对于客户端和服务端,在初始化 ChannelPipeline 时,增加 DelimiterBasedFrameDecoder 解码器,用于对以分隔符做码流结束标识的消息做自动解码,服务端代码示例如下(EventTriggerServer 类):

---

```
//代码省略  
.childHandler(new ChannelInitializer<SocketChannel>() {  
    @Override  
    public void initChannel(SocketChannel ch)  
        throws Exception {  
        ByteBuf delimiter = Unpooled.copiedBuffer("$_"  
            .getBytes());  
        ch.pipeline().addLast(  
            new DelimiterBasedFrameDecoder(2048,  
                delimiter));  
        ch.pipeline().addLast(new StringDecoder());  
        ch.pipeline().addLast(new EventTriggerServerHandler());  
    }  
});  
//代码省略
```

---

对上述测试用例做验证,测试结果如图 15-1 所示。

```
This is 1 times receive ReadComplete event.  
This is 2 times receive ReadComplete event.  
This is 3 times receive ReadComplete event.  
This is 4 times receive ReadComplete event.  
This is 5 times receive ReadComplete event.  
This is 6 times receive ReadComplete event.  
This is 7 times receive ReadComplete event.  
This is 8 times receive ReadComplete event.  
This is 9 times receive ReadComplete event.  
This is 1 times receive client : [Hi,welcome to Netty Hi,welcome to Netty Hi,welcome to Netty Hi,welcome to Netty Hi,  
Netty Hi,welcome to Netty ]  
This is 10 times receive ReadComplete event.
```

图 15-1 测试结果



上述测试结果表明, 对于 `channelRead` 方法, 如果前面添加了对应协议的解码器, 则只有在消息被解码成功后才会调用 `channelRead` 方法。而 `channelReadComplete` 方法的调用机制则不同, 只要底层的 `SocketChannel` 读到了 `ByteBuf`, 就会触发一次调用, 对于一个完整的业务消息, 可能会多次触发调用。

找到出现问题的原因之后, 将 `channelReadComplete` 方法中的业务逻辑移到 `channelRead` 方法中执行, 生产环境升级之后问题得到解决。

## 15.2 ChannelHandler 使用的一些误区总结

`ChannelHandler` 由 `ChannelPipeline` 触发, 业务经常使用的方法包括 `channelRead` 方法、`channelReadComplete` 方法和 `exceptionCaught` 方法等, 它的使用比较简单, 但是还是有一些容易出错的地方, 使用不当就会导致出现上一节中的问题。

### 15.2.1 channelReadComplete 方法调用

对于 `channelReadComplete` 方法的调用, 我们很容易误认为前面已经增加了对应协议的编解码器, 所以只有消息解码成功才会调用 `channelReadComplete` 方法。实际上它的调用与用户是否添加协议解码器无关, 只要对应的 `SocketChannel` 成功读到了 `ByteBuf`, 它就会被触发, 相关代码如下 (`NioByteUnsafe` 类):

---

```
public final void read() {  
    //代码省略  
    try {  
        do {  
            byteBuf = allocHandle.allocate(allocator);  
            allocHandle.lastBytesRead(doReadBytes(byteBuf));  
            if (allocHandle.lastBytesRead() <= 0) {  
                byteBuf.release();  
                byteBuf = null;  
                close = allocHandle.lastBytesRead() < 0;  
            }  
        }  
    }  
}
```



## Netty 进阶之路：跟着案例学 Netty

```
        if (close) {
            readPending = false;
        }
        break;
    }
    allocHandle.incMessagesRead(1);
    readPending = false;
    pipeline.fireChannelRead(byteBuf);
    byteBuf = null;
} while (allocHandle.continueReading());
allocHandle.readComplete();
pipeline.fireChannelReadComplete();
//代码省略
}
```

对于大部分协议解码器，例如 Netty 内置的 `ByteToMessageDecoder`，它会调用具体的协议解码器对 `ByteBuf` 解码，只有解码成功，才会调用后续 `ChannelHandler` 的 `channelRead` 方法，代码如下（`ByteToMessageDecoder` 类）：

```
static void fireChannelRead(ChannelHandlerContext ctx, CodecOutputList msgs,
int numElements) {
    for (int i = 0; i < numElements; i++) {
        ctx.fireChannelRead(msgs.getUnsafe(i));
    }
}
```

`channelReadComplete` 方法则属于透传调用，即无论是否有完整的消息被解码成功，只要读到消息，都会触发后续 `ChannelHandler` 的 `channelReadComplete` 方法调用，代码如下（`ByteToMessageDecoder` 类）：

```
public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
    numReads = 0;
    discardSomeReadBytes();
    if (decodeWasNull) {
```



```

        decodeWasNull = false;
        if (!ctx.channel().config().isAutoRead()) {
            ctx.read();
        }
    }
    ctx.fireChannelReadComplete();
}

```

---

## 15.2.2 ChannelHandler 职责链调用

ChannelPipeline 以链表的方式管理某个 Channel 对应的所有 ChannelHandler，需要说明的是，下一个 ChannelHandler 的触发需要在当前 ChannelHandler 中显式调用，而不是自动触发式调用，相关代码如下（SslHandler 类）：

```

public void channelActive(final ChannelHandlerContext ctx) throws Exception {
    if (!startTls) {
        startHandshakeProcessing();
    }
    ctx.fireChannelActive();
}

```

---

如果遗忘了调用 ctx.fireChannelActive 方法，则 SslHandler 后续的 ChannelHandler 的 channelActive 方法将不会被执行，职责链执行到 SslHandler 就会中断。

Netty 内置的 TailContext 有时候会执行一些系统性的清理操作，例如当 channelRead 方法执行完成，将请求消息（例如 ByteBuf）释放，防止因为业务遗漏释放而导致内存泄漏（在内存池模式下会导致内存泄漏），相关代码如下（TailContext 类）：

```

protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug(
            "Discarded inbound message {} that reached at the tail of
the pipeline. " +
            "Please check your pipeline configuration.", msg);
    }
}

```

```
    } finally {  
        ReferenceCountUtil.release(msg);  
    }  
}
```

---

当执行完业务的最后一个 `ChannelHandler` 时,要判断是否需要调用系统的 `TailContext`, 如果需要, 则通过 `ctx.firexxx` 方法调用。

## 15.3 总结

在通常情况下, 在做功能测试或者并发压力不大时, HTTP 请求消息可以一次性接收完成, 此时 `ChannelHandler` 的 `channelReadComplete` 方法会被调用一次, 但是当整个包消息经过多次读取才能完成解码时, `channelReadComplete` 方法就会被调用多次。如果业务的功能正确性依赖 `channelReadComplete` 方法的调用次数, 当客户端并发压力大或者采用 `chunked` 编码时, 功能就会出错。因此, 需要熟悉和掌握 Netty 的事件触发机制及 `ChannelHandler` 的调用策略, 这样才能防止在生产环境中“踩坑”。

## 第 16 章

---

# Netty 流量整形应用案例

当系统负载压力比较大时，系统进入过负荷状态，可能是 CPU、内存资源已经过载，也可能是应用进程内部的资源几乎耗尽，如果继续全量处理业务，可能会导致长时间的 Full GC、消息严重积压或者应用进程宕机，最终将压力转移到集群中的其他节点，引起级联故障。通过动态流控，拒绝一定比例新接入的请求消息，可以保障系统不被压垮。

除了动态流控，有时候还需要对消息的读取和发送速度做控制，以便消息能以较恒定的速度发送到下游网元，保护下游各系统不受突发的流量冲击，通过 Netty 提供的流量整形功能，就可以达到控制消息读取和发送速度的目标。

### 16.1 Netty 流量整形功能

---

大多数的商用系统都由多个网元或者部件组成，例如参与短信互动，会涉及手机、基站、短信中心、短信网关、SP/CP 等网元。不同网元或者部件的处理性能不同，为了防止因为大量业务或者下游网元性能低导致下游网元被压垮，有时候需要系统提供流量整形功能。

### 16.1.1 通用的流量整形功能简介

流量整形（Traffic Shaping）是一种主动调整流量输出速度的措施。一个典型的应用是基于下游网络节点的 TPS 指标控制本地流量的输出。流量整形与流量控制的主要区别在于，流量整形是对流量控制中需要丢弃的报文进行缓存——通常是将它们放入缓冲区或队列。当令牌桶有足够多的令牌时，再均匀地向外发送这些被缓存的报文。流量整形与流量控制的另一区别是，整形可能会增加延迟，而流控几乎不引入额外的延迟。

流量整形的工作原理如图 16-1 所示。

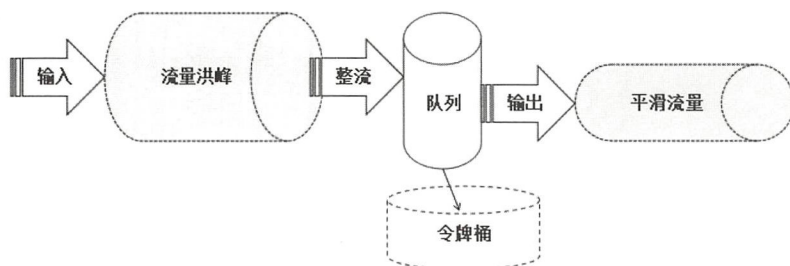


图 16-1 流量整形的工作原理

### 16.1.2 Netty 流量整形功能简介

Netty 内置了三种流量整形功能。

（1）单个链路的流量整形：ChannelTrafficShapingHandler，可以对某个链路的消息发送和读取速度进行控制。

（2）全局流量整形：GlobalTrafficShapingHandler，针对某个进程所有链路的消息发送和读取速度的总和进行控制。

（3）全局和单个链路综合型流量整形：GlobalChannelTrafficShapingHandler，同时对全局和单个链路的消息发送和读取速度进行控制。

Netty 流量整形的主要作用有两个。

（1）防止由于上、下游网元性能不均衡导致下游网元被压垮，业务流程中断。

(2) 防止由于通信模块接收消息过快，后端业务线程处理不及时，导致出现“撑死”问题。

## 16.2 Netty 流量整形应用

流量整形应用相对比较简单，只需要将流量整形 ChannelHandler 添加到业务解码器之前，即可对消息的读取和发送速度进行均匀控制，而且不会丢弃消息。下面以服务端对单个 Channel 的读取速度进行整形为例进行说明。

### 16.2.1 流量整形示例代码

创建客户端 TCP 连接之后，启动定时任务，以 10MB/s 的速度向服务端发送请求消息，示例代码如下（TrafficShappingClientHandler 类）：

---

```
public class TrafficShappingClientHandler extends
ChannelInboundHandlerAdapter {
    private static AtomicInteger SEQ = new AtomicInteger(0);
    static final byte[] ECHO_REQ = new byte[1024 * 1024];
    static final String DELIMITER = "$_";
    static ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        scheduledExecutorService.scheduleAtFixedRate(()
            -> {
                ByteBuf buf = null;
                for (int i = 0; i < 10; i++) {
                    buf = Unpooled.copiedBuffer(ECHO_REQ, DELIMITER.getBytes());
                    SEQ.getAndAdd(buf.readableBytes());
                    if (ctx.channel().isWritable())
                        ctx.write(buf);
                }
            }, 0, 1, TimeUnit.SECONDS);
    }
}
```



```

    }
    ctx.flush();
    int counter = SEQ.getAndSet(0);
    System.out.println("The client send rate is : " + counter + "
bytes/s");
    }, 0, 1000, TimeUnit.MILLISECONDS);
}
//代码省略
}

```

---

在服务端添加 ChannelTrafficShapingHandler 对消息读取速度进行整形，代码示例如下 (TrafficShappingServer):

---

```

static void fireChannelRead(ChannelHandlerContext ctx, CodecOutputList msgs,
    .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch)
            throws Exception {
            ch.pipeline().addLast("Channel Traffic Shaping",new
ChannelTrafficShapingHandler(1024 * 1024,1024 * 1024, 1000));
            ByteBuf delimiter = Unpooled.copiedBuffer("$_"
                .getBytes());
            ch.pipeline().addLast(
                new DelimiterBasedFrameDecoder(2048 * 1024,
                    delimiter));
            ch.pipeline().addLast(new StringDecoder());
            ch.pipeline().addLast(new TrafficShapingServerHandler());
        }
    });
//代码省略
}

```

---

在服务端的 ChannelHandler 中，启动定时任务对消息的读取速度进行打印，代码如下 (TrafficShapingServerHandler 类):

---

```

public class TrafficShapingServerHandler extends ChannelInboundHandlerAdapter {
    AtomicInteger counter = new AtomicInteger(0);
    static ScheduledExecutorService es = Executors.newScheduledThreadPool(1);
    public TrafficShapingServerHandler() {
        es.scheduleAtFixedRate(() ->
        {
            System.out.println("The server receive client rate is : " +
counter.getAndSet(0) + " bytes/s");
            }, 0, 1000, TimeUnit.MILLISECONDS);

        }
    }
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        String body = (String) msg;
        counter.addAndGet(body.getBytes().length);
        body += "$_";
        ByteBuf echo = Unpooled.copiedBuffer(body.getBytes());
        ctx.writeAndFlush(echo);
    }
}
//代码省略
}

```

---

## 16.2.2 流量整形功能测试

对代码进行测试，客户端运行结果如图 16-2 所示，客户端以约 10MB/s 的速度发送请求消息。

```

The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s
The client send rate is : 10485780 bytes/s

```

图 16-2 客户端运行结果

服务端测试结果，如图 16-3 所示，服务端以 1MB/s 的速度读取请求消息，通过添加 ChannelTrafficShapingHandler 实现了精准的流量整形。

```
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
```

图 16-3 流量整形服务端测试结果

## 16.3 Netty 流量整形工作机制

流量整形的工作原理：拦截 channelRead 和 write 方法，计算当前需要发送的消息大小，对读取和发送阈值进行判断，如果达到了阈值，则暂停读取和发送消息，待下一个周期继续处理，以实现在某个周期内对消息读写速度进行控制。

### 16.3.1 流量整形工作原理和源码分析

#### 1. 消息读取的流量整形

以 ChannelTrafficShapingHandler 为例，消息读取的流量整形工作流程如图 16-4 所示。

流程的关键步骤解读如下。

(1) 在解码之前拦截 channelRead 方法，计算读取的 ByteBuf 大小，源码如下 (AbstractTrafficShapingHandler 类)：

```
protected long calculateSize(Object msg) {
    if (msg instanceof ByteBuf) {
        return ((ByteBuf) msg).readableBytes();
    }
    if (msg instanceof ByteBufHolder) {

```

```

        return ((ByteBufHolder) msg).content().readableBytes();
    }
    return -1;
}

```

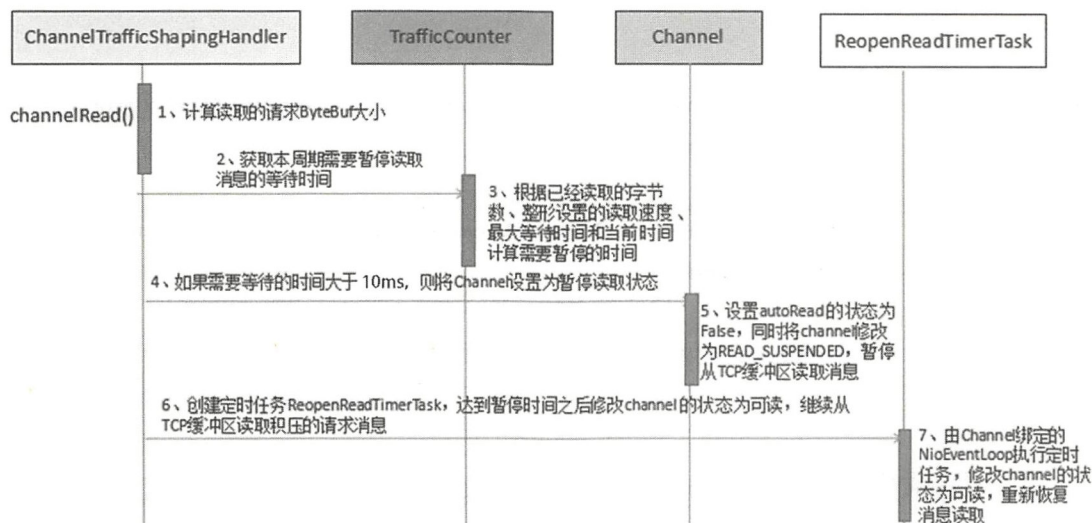


图 16-4 消息读取的流量整形工作流程

(2) 计算需要暂停读取消息的等待时间, 代码如下 (TrafficCounter 类):

```

public long readTimeToWait(final long size, final long limitTraffic, final
long maxTime, final long now) {
    bytesRecvFlowControl(size);
    if (size == 0 || limitTraffic == 0) {
        return 0;
    }
    final long lastTimeCheck = lastTime.get();
    long sum = currentReadBytes.get();
    long localReadingTime = readingTime;
    long lastRB = lastReadBytes;
    final long interval = now - lastTimeCheck;
    long pastDelay = Math.max(lastReadingTime - lastTimeCheck, 0);

```

```

        if (interval > AbstractTrafficShapingHandler.MINIMAL_WAIT) {
            long time = sum * 1000 / limitTraffic - interval + pastDelay;
            if (time > AbstractTrafficShapingHandler.MINIMAL_WAIT) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Time: " + time + ':' + sum + ':' + interval
+ ':' + pastDelay);
                }
                if (time > maxTime && now + time - localReadingTime > maxTime) {
                    time = maxTime;
                }
                readingTime = Math.max(localReadingTime, now + time);
                return time;
            }
            readingTime = Math.max(localReadingTime, now);
            return 0;
        }
        //代码省略
    }

```

---

(3) 满足整形条件，则修改 Channel 的状态为非自动读取，并将 READ\_SUSPENDED 的属性修改为 True，Channel 进入整形状态，不再从 TCP 缓冲区读取请求消息，相关代码如下（AbstractTrafficShapingHandler 类 channelRead 方法）：

```

{
    //代码省略
    if (config.isAutoRead() && isHandlerActive(ctx)) {
        config.setAutoRead(false);
        channel.attr(READ_SUSPENDED).set(true);
    }
    //代码省略
}

```

---

(4) 创建恢复 Channel 为可读的定时任务，由 Channel 对应的 NioEventLoop 执行，代码如下（AbstractTrafficShapingHandler 类 channelRead 方法）：



---

```

{
    //代码省略
    Attribute<Runnable> attr = channel.attr(REOPEN_TASK);
    Runnable reopenTask = attr.get();
    if (reopenTask == null) {
        reopenTask = new ReopenReadTimerTask(ctx);
        attr.set(reopenTask);
    }
    ctx.executor().schedule(reopenTask, wait, TimeUnit.MILLISECONDS);
    //代码省略
}

```

---

(5)到达暂停读取时间之后,触发定时任务,重新修改 Channel 的 READ\_SUSPENDED 属性为 False,同时将 autoRead 设置为 True,代码如下 (ReopenReadTimerTask 类):

---

```

public void run() {
    Channel channel = ctx.channel();
    ChannelConfig config = channel.config();
    if (!config.isAutoRead() && isHandlerActive(ctx)) {
        if (logger.isDebugEnabled()) {
            logger.debug("Not unsuspend: " + config.isAutoRead() + ':' +
                isHandlerActive(ctx));
        }
        channel.attr(READ_SUSPENDED).set(false);
    } else {
        //代码省略
    }
    channel.attr(READ_SUSPENDED).set(false);
    config.setAutoRead(true);
    channel.read();
}
//代码省略
}

```

---

## 2. 消息发送的流量整形

消息发送的流量整形工作流程如图 16-5 所示。

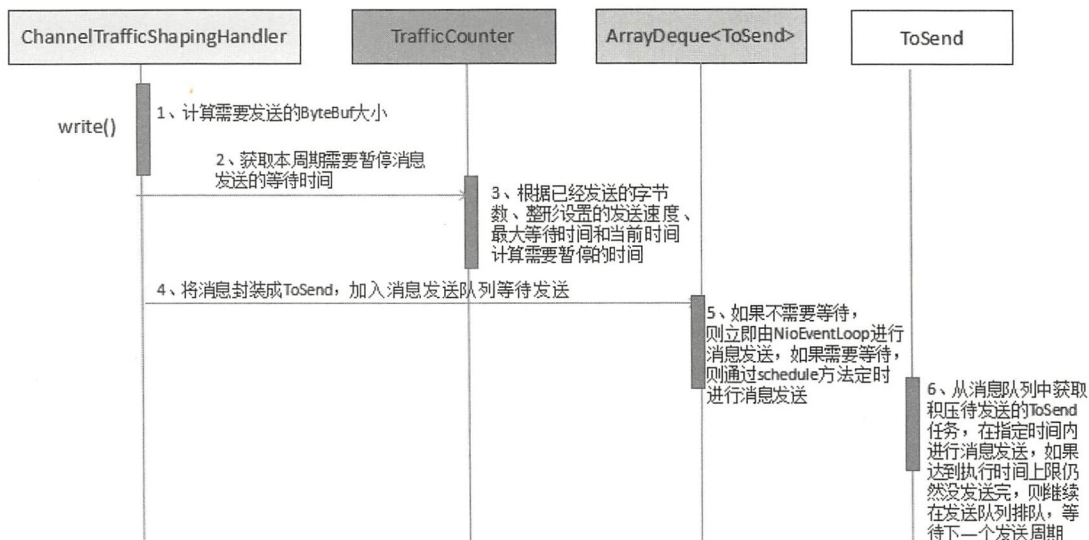


图 16-5 消息发送的流量整形工作流程

流程的关键步骤解读如下。

(1) 计算需要暂停发送的等待时间，代码如下（TrafficCounter 类）：

```

public long writeTimeToWait(final long size, final long limitTraffic, final
long maxTime, final long now) {
    bytesWriteFlowControl(size);
    if (size == 0 || limitTraffic == 0) {
        return 0;
    }
    final long lastTimeCheck = lastTime.get();
    long sum = currentWrittenBytes.get();
    long lastWB = lastWrittenBytes;
    long localWritingTime = writingTime;
    long pastDelay = Math.max(lastWritingTime - lastTimeCheck, 0);
    final long interval = now - lastTimeCheck;

```

```

        if (interval > AbstractTrafficShapingHandler.MINIMAL_WAIT) {
            long time = sum * 1000 / limitTraffic - interval + pastDelay;
            if (time > AbstractTrafficShapingHandler.MINIMAL_WAIT) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Time: " + time + ':' + sum + ':' + interval
+ ':' + pastDelay);
                }
                if (time > maxTime && now + time - localWritingTime > maxTime) {
                    time = maxTime;
                }
                writingTime = Math.max(localWritingTime, now + time);
                return time;
            }
            writingTime = Math.max(localWritingTime, now);
            return 0;
        }
        //代码省略
    }

```

(2) 如果需要暂停发送时间大于 10ms, 则通过定时任务进行消息发送, 否则直接封装成 ToSend 任务, 添加到 NioEventLoop 的 Task 队列立即执行, 代码如下 (AbstractTrafficShapingHandler 类):

```

    public void write(final ChannelHandlerContext ctx, final Object msg, final
ChannelPromise promise)
        throws Exception {
        long size = calculateSize(msg);
        long now = TrafficCounter.millisecondFromNano();
        if (size > 0) {
            long wait = trafficCounter.writeTimeToWait(size, writeLimit,
maxTime, now);
            if (wait >= MINIMAL_WAIT) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Write suspend: " + wait + ':' + ctx.channel().

```

```

config().isAutoRead() + ':'
        + isHandlerActive(ctx));
    }
    submitWrite(ctx, msg, size, wait, now, promise);
    return;
}
}
submitWrite(ctx, msg, size, 0, now, promise);
}
}

```

---

(3) 由 Channel 对应的 NioEventLoop 从 messagesQueue 中循环获取待发送的 ToSend 任务，执行消息发送操作，代码如下（ChannelTrafficShapingHandler 类）：

---

```

private void sendAllValid(final ChannelHandlerContext ctx, final long now) {
    synchronized (this) {
        ToSend newToSend = messagesQueue.pollFirst();
        for (; newToSend != null; newToSend = messagesQueue.pollFirst()) {
            if (newToSend.relativeTimeAction <= now) {
                long size = calculateSize(newToSend.toSend);
                trafficCounter.bytesRealWriteFlowControl(size);
                queueSize -= size;
                ctx.write(newToSend.toSend, newToSend.promise);
            } else {
                messagesQueue.addFirst(newToSend);
                break;
            }
        }
        if (messagesQueue.isEmpty()) {
            releaseWriteSuspended(ctx);
        }
    }
    ctx.flush();
}

```

---

需要注意的是，如果到达了执行时间，之前取出来待发送的消息需要重新加入消息发送队列，由于消息是有先后顺序的，所以重新加入队列首位（addFirst）。

## 16.3.2 并发编程在流量整形中的应用

作为异步事件驱动、高性能的 NIO 框架，Netty 大量运用了 Java 多线程编程技巧。并发编程处理得恰当与否，直接影响架构的性能。下面看一下流量整形中使用的并发编程技巧。

### 1. volatile 的使用

关键字 volatile 是 Java 提供的最轻量级的同步机制，Java 内存模型对 volatile 专门定义了一些特殊的访问规则，当一个变量被 volatile 修饰后，它将具备以下两种特性。

（1）线程可见性：一个线程修改了被 volatile 修饰的变量后，无论是否加锁，其他线程都可以立即看到最新的修改，而普通变量却做不到这一点。

（2）禁止指令重排序优化：普通的变量仅保证在方法的执行过程中所有依赖赋值结果的地方都能获取正确的结果，而不能保证变量赋值操作的顺序与程序代码的执行顺序一致。

以 maxWriteSize 为例，它被声明为 volatile 型的变量，因为它提供了 public 的 set 方法，用户线程或者其他 Channel 对应的 NioEventLoop 线程可以调用它修改 maxWriteSize 的值，设置 maxWriteSize 属性的接口定义如图 16-6 所示。

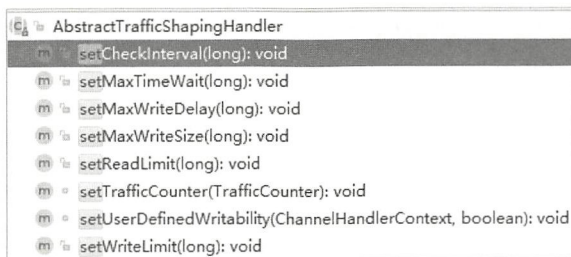


图 16-6 设置 maxWriteSize 属性的接口定义

在 Channel 绑定的 NioEventLoop 线程中需要读取 maxWriteSize，如果它不是 volatile 型的变量，则业务线程对 maxWriteSize 修改后 NioEventLoop 可能会读取到脏数据，代码如下（AbstractTrafficShapingHandler 类）：



---

```

private void sendAllValid(final ChannelHandlerContext ctx, final long now) {
    void checkWriteSuspend(ChannelHandlerContext ctx, long delay, long
queueSize) {
        if (queueSize > maxWriteSize || delay > maxWriteDelay) {
            setUserDefinedWritability(ctx, false);
        }
        //代码省略
    }
}

```

---

## 2. 减小锁的范围

由于涉及多线程调用，需要对消息发送队列 `ArrayDeque<ToSend>` 加锁，但是不需要对后续的其他操作加锁，例如通过 `NioEventLoop` 的 `schedule` 方法执行定时任务，因为它本身就是并发安全的，所以不需要额外加锁，相关代码如下（`ChannelTrafficShapingHandler` 类）：

---

```

void submitWrite(final ChannelHandlerContext ctx, final Object msg,
    final long size, final long delay, final long now,
    final ChannelPromise promise) {
    final ToSend newToSend;
    synchronized (this) {
        if (delay == 0 && messagesQueue.isEmpty()) {
            trafficCounter.bytesRealWriteFlowControl(size);
            ctx.write(msg, promise);
            return;
        }
        //代码省略
    }
    final long futureNow = newToSend.relativeTimeAction;
    ctx.executor().schedule(new Runnable() {
        @Override
        public void run() {
            sendAllValid(ctx, futureNow);
        }
    })
}

```

---

```

    }, delay, TimeUnit.MILLISECONDS);
}
}

```

### 3. 原子类

由于支持对所有的 Channel 做流量整形，不同的 Channel 会绑定不同的 NioEventLoop 线程，所以消息发送和读取的计算都是并发操作的，如果不做同步保护，统计数据将不准确。

如果使用同步关键字，最主要的问题就是进行线程阻塞和唤醒带来的性能的额外损耗，因此这种同步被称为阻塞同步，它属于一种悲观的并发策略，被称为悲观锁。随着硬件和操作系统指令集的发展，产生了非阻塞同步，被称为乐观锁。简单地说，就是先进行操作，操作完成再判断操作是否成功，是否有并发问题，如果有则进行失败补偿，如果没有就算操作成功，这样就从根本上避免了同步锁的弊端。

目前，在 Java 中应用最广泛的非阻塞同步是 CAS，在 IA64、X86 指令集中通过 `cmpxchg` 指令完成 CAS 功能，在 `sparc-TSO` 中由 `case` 指令完成，在 ARM 和 PowerPC 架构下，需要使用一对 `ldrex/strex` 指令完成。从 JDK 1.5 以后，可以使用 CAS 操作，该操作由 `sun.misc.Unsafe` 类的 `compareAndSwapInt()` 和 `compareAndSwapLong()` 等方法包装提供。在通常情况下 `sun.misc.Unsafe` 类对于开发者是不可见的，因此 JDK 提供了很多 CAS 包装类简化开发者的工作，如 `AtomicInteger`。

在流量整形中大量使用了原子类提升并发操作的安全性和性能，代码如下（`TrafficCounter` 类）：

```

{
//代码省略
private final AtomicLong currentWrittenBytes = new AtomicLong();
/**
 * Current read bytes
 */
private final AtomicLong currentReadBytes = new AtomicLong();
/**
 * Long life written bytes

```

```
    */  
    private final AtomicLong cumulativeWrittenBytes = new AtomicLong();  
    /**  
     * Long life read bytes  
     */  
    private final AtomicLong cumulativeReadBytes = new AtomicLong();  
    //代码省略  
}
```

---

### 16.3.3 使用流量整形的一些注意事项总结

---

#### 1. 流量整形 ChannelHandler 添加位置

因为需要计算请求和发送消息的大小,消息类型必须是 ByteBuf 或者 ByteBufHolder, 所以流量整形 ChannelHandler 需要添加到业务编码之后、解码之前, 代码示例如下 (TrafficShappingServer 类):

---

```
{  
    //代码省略  
    .childHandler(new ChannelInitializer<SocketChannel>() {  
        @Override  
        public void initChannel(SocketChannel ch)  
            throws Exception {  
            ch.pipeline().addLast("Channel Traffic Shaping", new  
ChannelTrafficShapingHandler(1024 * 1024, 1024 * 1024, 1000));  
            ByteBuf delimiter = Unpooled.copiedBuffer("$_  
                .getBytes());  
            ch.pipeline().addLast(  
                new DelimiterBasedFrameDecoder(2048 * 1024,  
                    delimiter));  
            ch.pipeline().addLast(new StringDecoder());  
            ch.pipeline().addLast(new TrafficShappingServerHandler());  
        }  
    })  
}
```

```
//代码省略
```

```
}
```

## 2. 全局流量整形实例只需要创建一次

全局流量整形 `GlobalChannelTrafficShapingHandler` 和 `GlobalTrafficShapingHandler` 是全局共享的, 因此实例只需要创建一次, 添加到不同的 `ChannelPipeline` 即可, 不要创建多个实例, 否则流量整形将失效。 `GlobalTrafficShapingHandler` 类定义如图 16-7 所示。

```
@Sharable
public class GlobalTrafficShapingHandler extends AbstractTrafficShapingHandler {
```

图 16-7 `GlobalTrafficShapingHandler` 类定义

## 3. 流量整形参数调整不要过于频繁

虽然通过 `AbstractTrafficShapingHandler` 的 `configure` 接口可以动态调整流量整形的读写速度和检测周期, 但是由于调整之后需要对一些统计数据进行重新设置和重新计算, 而且在下一个周期才能生效, 过于频繁的参数调整会导致流量整形不精确, 甚至失效。动态调整流量整形参数的相关接口如图 16-8 所示。

```
AbstractTrafficShapingHandler
m configure(long): void
m configure(long, long): void
m configure(long, long, long): void
```

图 16-8 动态调整流量整形参数的相关接口

## 4. 资源释放问题

在 `Channel` 关闭或者流量整形 `ChannelHandler` 被移除时, 由于 `ChannelTrafficShapingHandler` 持有消息发送队列, 如果不对消息队列进行清空处理, 则会导致待发送消息丢失, 或者消息队列积压, 引起内存泄漏 (频繁地断连和重连, 会创建  $N$  个 `ChannelTrafficShapingHandler` 实例, 对应  $N$  个消息发送队列)。

Netty 框架已经考虑了上述场景, 当连接关闭时, 会调用 `handlerRemoved` 方法, 将待发送的消息全部释放, 防止内存泄漏, 代码如下 (`ChannelTrafficShapingHandler` 类):

```
public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
    trafficCounter.stop();
}
```

```

synchronized (this) {
    if (ctx.channel().isActive()) {
        //代码省略
    } else {
        for (ToSend toSend : messagesQueue) {
            if (toSend.toSend instanceof ByteBuf) {
                ((ByteBuf) toSend.toSend).release();
            }
        }
        messagesQueue.clear();
    }
    releaseWriteSuspended(ctx);
    releaseReadSuspended(ctx);
    super.handlerRemoved(ctx);
}

```

---

如果连接正常，用户主动调用 `handlerRemoved` 删除流量整形 `ChannelHandler`，则将积压的消息全部发送完成，清空消息发送队列。由于消息发送成功后由 Netty 负责释放 `ByteBuf`，因此避免了内存泄漏，相关代码如下：

---

```

public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
    trafficCounter.stop();
    synchronized (this) {
        if (ctx.channel().isActive()) {
            for (ToSend toSend : messagesQueue) {
                long size = calculateSize(toSend.toSend);
                trafficCounter.bytesRealWriteFlowControl(size);
                queueSize -= size;
                ctx.write(toSend.toSend, toSend.promise);
            }
        } else {
            //代码省略...
        }
    }
}

```



```

        messagesQueue.clear();
    }
    releaseWriteSuspended(ctx);
    releaseReadSuspended(ctx);
    super.handlerRemoved(ctx);
}

```

## 5. 消息发送保护机制

通过流量整形可以控制发送速度，但是它的控制原理是将待发送的消息封装成 Task 放入消息队列，等待执行时间到达后继续发送，所以如果业务发送线程不判断 Channel 的可写状态，就可能会导致 OOM 等问题。将 16.2 中的代码进行改造，注释掉 TrafficShappingClientHandler 的 Channel 可写状态判断，代码如下：

```

public void channelActive(ChannelHandlerContext ctx) {
    scheduledExecutorService.scheduleAtFixedRate(()
        -> {
            ByteBuf buf = null;
            for (int i = 0; i < 10; i++) {
                buf = Unpooled.copiedBuffer(ECHO_REQ, DELIMITER.getBytes());
                SEQ.getAndAdd(buf.readableBytes());
                // if (ctx.channel().isWritable())
                ctx.write(buf);
            }
        }
    );
    //代码省略
}

```

运行一段时间之后，客户端就会抛出异常。

服务端读取的请求消息速度为 0，如图 16-9 所示。

```

The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 1048576 bytes/s
The server receive client rate is : 0 bytes/s
The server receive client rate is : 0 bytes/s
The server receive client rate is : 0 bytes/s

```

图 16-9 服务端读取的请求消息速度为 0

客户端发生了 OOM 异常，如图 16-10 所示。

```
io.netty.util.internal.OutOfDirectMemoryError: failed to allocate 16777216 byte(s) of direct memory (used: 905969664, max: 913833984)
    at io.netty.util.internal.PlatformDependent.incrementMemoryCounter(PlatformDependent.java:640)
    at io.netty.util.internal.PlatformDependent.allocateDirectNoCleaner(PlatformDependent.java:594)
    at io.netty.buffer.PoolArena$DirectArena.allocateDirect(PoolArena.java:764)
    at io.netty.buffer.PoolArena$DirectArena.newChunk(PoolArena.java:740)
    at io.netty.buffer.PoolArena.allocateNormal(PoolArena.java:244)
    at io.netty.buffer.PoolArena.allocate(PoolArena.java:226)
    at io.netty.buffer.PoolArena.reallocate(PoolArena.java:391)
    at io.netty.buffer.PooledByteBuf.capacity(PooledByteBuf.java:118)
```

图 16-10 客户端发生了 OOM 异常

## 16.4 总结

流量整形与流控的最大区别在于，流控会拒绝消息，流量整形不拒绝和丢弃消息，无论接收量多大，它总能以近似恒定的速度下发消息，跟变压器的原理和功能类似。

## 第 17 章

---

# Netty SSL 应用案例

作为一个高性能的 NIO 通信框架，基于 Netty 的行业应用非常广泛，不同的行业、不同的应用场景，面临的安全挑战也不同。例如基于 Netty 构建 API Gateway，需要支持 HTTPS，此时就要用到 Netty 的 SSL 相关功能。

Netty 通过 JDK 的 `SSLEngine`，以 `SslHandler` 的方式提供对 SSL/TLS 安全传输的支持，极大地简化了用户的开发工作，降低了开发难度。考虑到性能因素，除了支持原生的 `JDK SSLEngine`，Netty 还提供了对 `OpenSSL` 的支持。尽管使用 Netty 开发 SSL/TLS 程序相对比较简单，但是在实际应用中很容易犯错，本章对一些常见的错误用法进行归纳总结，避免大家犯同样的错。

### 17.1 Netty SSL 功能简介

---

Netty 通过 `SslHandler` 提供了对 SSL 的支持，它支持的 SSL 协议类型包括 `SSL v2`、`SSL v3`、`TLS v1`、`TLS v1.1` 和 `TLS v1.2`。

如果使用 Netty 做 RPC 框架或者私有协议栈，RPC 框架面向非授信的第三方开放，例如将内部的一些能力通过服务对外开放，就需要进行安全认证。如果开放的是公网 IP，对

于一些安全性要求非常高的服务，例如在线支付、订购等，需要通过 SSL/TLS 进行通信。内部服务开放给外部非授信域示例如图 17-1 所示。

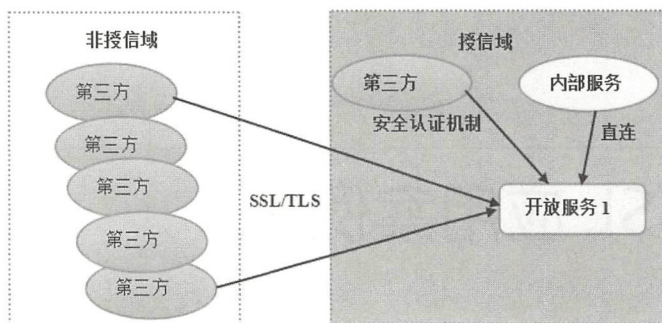


图 17-1 内部服务开放给外部非授信域示例

## 17.1.1 SSL 安全特性

### 1. 单向认证

单向认证即客户端验证服务端的合法性，服务端不验证客户端，它的工作原理如下。

(1) SSL 客户端向服务端传送客户端 SSL 协议的版本号、支持的加密算法种类、产生的随机数及其他可选信息。

(2) 服务端返回握手应答，向客户端传送确认 SSL 协议的版本号、加密算法的种类、随机数及其他相关信息。

(3) 服务端向客户端发送自己的公钥。

(4) 客户端对服务端的证书进行认证，服务端的合法性校验包括证书是否过期、发行服务器证书的 CA 是否可靠、发行者证书的公钥能否正确解开服务器证书的“发行者的数字签名”、服务器证书上的域名是否与服务器的实际域名匹配等。

(5) 客户端随机产生一个用于后面通信的“对称密码”，然后用服务端的公钥对其加密，将加密后的“预备主密码”传给服务端。

(6) 服务端将用自己的私钥解开加密的“预备主密码”，然后通过一系列步骤来产生主密码。



(7) 客户端向服务端发出信息，指明后面的数据通信将使用主密码作为对称密钥，同时通知服务器客户端的握手过程结束。

(8) 服务端向客户端发出信息，指明后面的数据通信将使用主密码作为对称密钥，同时通知客户端服务器端的握手过程结束。

SSL 的握手部分结束，SSL 安全通道建立，客户端和服务端开始使用相同的对称密钥对数据进行加密，然后通过 Socket 进行传输。

## 2. 双向认证

SSL 双向认证相比单向认证多了一步，即服务端发送认证请求消息给客户端，客户端发送自签名证书给服务端进行安全认证。

## 3. CA 认证

使用 JDK keytool 工具生成的数字证书是自签名的，自签名就是指证书只能保证自己是完整且没有经过非法修改的，但是无法保证这个证书是属于谁的。为了对自签名证书进行认证，需要每个客户端和服务端都交换自己自签名的私有证书，对于一个大型网站或者应用服务器来说，工作量是非常大的。

基于自签名的 SSL 双向认证，只要客户端或者服务端修改了密钥和证书，就需要重新进行签名和证书交换，这种调试和维护工作量是非常大的。因此，在实际的商用系统中往往会通过第三方 CA 证书颁发机构进行签名和验证。例如浏览器就保存了几个常用的 CA\_ROOT，每次连接到网站时只要这个网站的证书是经过这些 CA\_ROOT 签名的就可以通过验证。

### 17.1.2 Netty SSL 实现机制

Netty 提供了两种 SSL 实现机制。

(1) JDK SSL: JdkSslEngine，继承自 JDK SSLEngine，对 JDK 原生的 SSL 功能做了包装和增强。

(2) OpenSSL: OpenSslEngine，通过原生的方式调用 OpenSSL，提升性能。





通过 `-Dio.netty.handler.ssl.noOpenSsl` 可以显式开启或者禁用 `OpenSsl`，对于性能要求比较高的应用，建议使用 `OpenSsl` 替换 `JDK` 的 `SslEngine`。

## 17.2 Netty 客户端 SSL 握手超时问题

业务基于 `Netty` 开发了服务端和客户端，采用 `TCP` 私有协议通信，业务运行正常。后来按照公司安全规范进行升级，双方采用 `TLS` 进行通信，升级之后客户端超时。

### 17.2.1 握手超时原因定位

客户端通过 `TLS` 连接服务端后发送请求消息，没有响应，首先考虑双方的 `SSL` 握手是否成功，可以通过抓包或者打印 `SSL` 调测日志的方式查看双方的 `SSL` 握手过程，在客户端开启 `SSL` 调测相关参数，如图 17-2 所示。

VM options:	-Djavax.net.debug=ssl,handshake,data,trustmanager
-------------	---

图 17-2 配置 `SSL` 调测日志

重启客户端，查看握手日志，发现客户端发送完 “\*\*\* ClientHello, TLSv1.2” 之后，没有收到服务端的响应，如图 17-3 所示。

```
*** ClientHello, TLSv1.2
RandomCookie: GMT: 1519305926 bytes = { 63, 111, 119, 138, 221, 122, 173, 69,
Session ID: {}
Cipher Suites: [TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256, TLS_ECDHE_RSA_WITH_AES
Compression Methods: { 0 }
Extension elliptic_curves, curve names: {secp256r1, secp384r1, secp521r1, sect2
Extension ec_point_formats, formats: [uncompressed]
Extension signature_algorithms, signature_algorithms: SHA512withECDSA, SHA512w
***
[write] MD5 and SHA1 hashes: len = 161
0000: 01 00 00 9D 03 03 5B 8F C5 C6 3F 6F 77 8A DD 7A .....[...?ow..z
0010: AD 45 26 59 E0 43 30 A7 B3 BC E7 F3 2C 5F 99 00 ..EsY.CO....._.
0020: 1B 43 11 AE C4 00 00 00 3A C0 23 C0 27 00 3C C0 .C.....#.(<.
0030: 25 C0 29 00 67 00 40 C0 09 C0 13 00 2F C0 04 C0 %).g.@...../...
0040: 0E 00 33 00 32 C0 2B C0 2F 00 9C C0 2D C0 31 00 ..3.2.+./...-..1.
0050: 9E 00 A2 C0 08 C0 12 00 0A C0 03 C0 0D 00 16 00 .....
0060: 13 00 FF 01 00 00 3A 00 0A 00 16 00 14 00 17 00 .....
0070: 18 00 19 00 09 00 0A 00 0B 00 0C 00 0D 00 0E 00 .....
0080: 16 00 0B 00 02 01 00 00 0D 00 16 00 14 06 03 06 .....
0090: 01 05 03 05 01 04 03 04 01 04 02 02 03 02 01 02 .....
00A0: 02
nioEventLoopGroup-2-1, WRITE: TLSv1.2 Handshake, length = 161
nioEventLoopGroup-2-1, called closeOutbound()
nioEventLoopGroup-2-1, closeOutboundInternal()
nioEventLoopGroup-2-1, SEND TLSv1.2 ALERT: warning, description = close_notify
```

图 17-3 客户端 `SSL` 握手超时日志



让服务端协助排查 TLS 握手没有返回应答的原因，经过代码检查发现服务端添加 SslHandler 的位置有问题，将它添加到了业务解码器后面，这样 SSL 握手消息首先会被业务解码器调用，不满足业务解码器的解析条件，握手消息不会调用后面的 SslHandler，所以导致握手失败，错误的代码示例如下：

```
if (SSLMODE.CSA.toString().equals(tlsMode))
    engine.setNeedClientAuth(true);
pipeline.addLast("framer", new DelimiterBasedFrameDecoder(8192,
    Delimiters.lineDelimiter()));
pipeline.addLast("ssl", new SslHandler(engine));
pipeline.addLast("decoder", new StringDecoder());
pipeline.addLast("encoder", new StringEncoder());
}
```

对代码进行修正，将 SslHandler 放到 ChannelPipeline 的首位（示例代码中 DelimiterBasedFrameDecoder 之前），重新验证，发现 SSL 握手成功。SSL 握手成功日志如图 17-4 所示。

```
*** Finished
verify_data: { 69, 116, 81, 17, 30, 155, 39, 114, 61, 122, 86, 136 }
***
[write] MD5 and SHA1 hashes: len = 16
0000: 14 00 00 0C 45 74 51 11 1E 9B 27 72 3D 7A 56 88 ...EtQ...r=zV.
nioEventLoopGroup-2-1, WRITE: TLSv1.2 Handshake, length = 80
nioEventLoopGroup-2-1, READ: TLSv1.2 Change Cipher Spec, length = 1
nioEventLoopGroup-2-1, READ: TLSv1.2 Handshake, length = 80
*** Finished
verify_data: { 142, 91, 34, 193, 212, 174, 9, 117, 88, 138, 35, 250 }
***
%% Cached client session: [Session-1, TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256]
[read] MD5 and SHA1 hashes: len = 16
0000: 14 00 00 0C 8E 5B 22 C1 D4 AE 09 75 58 8A 23 FA .....["....uX.#.
Welcome to Lilinfeng secure chat service!
Your session is protected by TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 cipher suite.
```

图 17-4 SSL 握手成功日志

## 17.2.2 Netty SSL 握手问题定位技巧

如果使用 JDK 的 SSL 引擎，则可以通过在 VM 启动参数中添加“-Djavax.net.debug=ssl,





handshake,data,trustmanager”的方式来打印 SSL 握手日志。如果使用 OpenSSL 引擎，则可以通过抓包的方式定位问题。

常见的 SSL 握手失败原因如下。

- (1) 证书不匹配，证书校验失败。
- (2) 证书过期。
- (3) 双方 SSL 引擎支持的 TLS/SSL 版本不一致，有些 SSL 引擎不支持高版本的 TLS/SSL。
- (4) 域名校验失败。
- (5) 双方支持的加密算法不一致，导致算法套件匹配失败。

## 17.3 SSL 握手性能问题

服务端采用 Netty 构建，在业务高峰期会出现客户端批量超时问题，超时之后有大量客户端使用 SSL 接入，导致服务端 CPU 使用率较高，部分客户端接入超时。

### 17.3.1 SSL 握手性能热点分析

模拟生产环境进行测试，对服务端进行 SSL 接入压测，模拟代码示例如下：

```
for (int i = 0; i < 10000; i++)  
{  
    ch = b.connect(host, port).channel();  
    TimeUnit.MILLISECONDS.sleep(1);  
}
```

对 Netty SSL 服务端 CPU 热点方法进行采集，服务端 SSL 握手热点方法快照如图 17-5 所示。





热点 - 方法	自用...
<pre>com.phei.netty.ssl.SecureChatSslContextFactory.getServerContext () com.phei.netty.ssl.SecureChatServerInitializer.initChannel () io.netty.channel.nio.NioEventLoop.run () io.netty.util.concurrent.SingleThreadEventExecutor\$5.run () io.netty.channel.ChannelHandlerInvokerUtil.invokeChannelRegisteredNow () io.netty.channel.ChannelInitializer.channelRegistered () io.netty.channel.DefaultChannelPipeline.fireChannelRegistered () io.netty.channel.DefaultChannelHandlerContext.fireChannelRegistered () io.netty.channel.DefaultChannelHandlerInvoker.invokeChannelRegistered () io.netty.channel.AbstractChannel\$AbstractUnsafe.access\$100 () io.netty.channel.AbstractChannel\$AbstractUnsafe.register0 ()</pre>	

图 17-5 服务端 SSL 握手热点方法快照

通过对 SecureChatSslContextFactory 的 getServerContext 方法进行分析,发现存在如下优化点:

```
KeyManagerFactory kmf = null;
if (pkPath != null) {
    KeyStore ks = KeyStore.getInstance("JKS");
    in = new FileInputStream(pkPath);
    ks.load(in, "sNetty".toCharArray());
    kmf = KeyManagerFactory.getInstance("SunX509");
    kmf.init(ks, "sNetty".toCharArray());
}
```

因为 keystore 证书通常不会变更,每当有新的 SSL 客户端接入就要重新加载证书,并发量高时会影响性能。因此可以把证书加载的文件流做成缓存来提高加载速度。

### 17.3.2 缓存和对象池

对于创建和销毁成本较高的对象或者经常使用的配置数据,将其保存到缓存或者对象池中,防止每次业务调用都实时创建和加载,可以极大地提升系统的性能。考虑到配置或者对象变更,本地缓存一般都要有变更通知和缓存失效机制,以保证数据的准确性。





## 17.4 SSL 事件监听机制

在一些场景下，业务需要了解底层 SSL 握手情况，以便进行相应业务逻辑处理，例如，针对 SSL 的连接数流控，需要在 SSL 握手成功之后对计数器进行累加操作。

### 17.4.1 握手成功事件

通过监听 `SslHandshakeCompletionEvent` 事件，可以在 SSL 握手成功后做业务逻辑处理，Netty 在握手成功时会触发该事件的调用，代码如下（`SslHandler` 类）：

```
private void setHandshakeSuccess() {
    handshakePromise.trySuccess(ctx.channel());
    if (logger.isDebugEnabled()) {
        logger.debug("{} HANDSHAKEN: {}", ctx.channel(),
engine.getSession().getCipherSuite());
    }
    ctx.fireUserEventTriggered(SslHandshakeCompletionEvent.SUCCESS);
    if (readDuringHandshake && !ctx.channel().config().isAutoRead()) {
        readDuringHandshake = false;
        ctx.read();
    }
}
```

业务 `ChannelHandler` 通过自定义 `userEventTriggered` 方法拦截并处理 `SslHandshakeCompletionEvent` 事件，即可实现握手成功之后的业务逻辑。

### 17.4.2 SSL 连接关闭事件

当 SSL 连接关闭时，会触发 `SslCloseCompletionEvent` 事件，相关代码如下（`SslHandler` 类）：

```
private void notifyClosePromise(Throwable cause) {
    if (cause == null) {
```







```
        if (sslClosePromise.trySuccess(ctx.channel())) {
            ctx.fireUserEventTriggered(SslCloseCompletionEvent.SUCCESS);
        }
    } else {
        if (sslClosePromise.tryFailure(cause)) {
            ctx.fireUserEventTriggered(new SslCloseCompletionEvent(cause));
        }
    }
}
```

用户通过 `userEventTriggered` 方法拦截并处理 `SslCloseCompletionEvent` 事件，实现定制业务逻辑的处理。

## 17.5 总结

当存在跨网络边界的 RPC 调用时，往往需要通过 TLS/SSL 对传输通道进行加密，以防止请求和响应消息中的敏感数据泄露。通过 Netty 提供的 `SslHandler` 可以方便地实现单向和双向认证，对于有较高性能诉求的场景，可以使用 OpenSSL 引擎来提升 SSL/TLS 的通信性能。





## 第 18 章

---

# Netty HTTPS 服务端高并发宕机案例

Netty 提供了原生的 HTTPS 协议栈，用户可以基于该协议栈开发各种 HTTPS 应用，在互联网的高并发场景下，如果客户端并发量过大，而服务端又没有针对并发连接数进行流控，或者没有弹性伸缩能力，则很有可能宕机。

HTTPS 服务端的性能优化，既有功能层面的小优化，又有架构层面的大优化，本章针对高并发场景下 HTTPS 服务端的可靠性保护和性能优化进行详细讲解。

## 18.1 Netty HTTPS 服务端宕机问题

---

某电商系统，内部两个模块之间采用 HTTPS 通信，平时业务运行正常，某天客户端突然发现大量超时，持续一段时间之后，发送给服务端的消息全部失败，吞吐量降为 0。查看服务端日志，发现大量 OOM 异常，初步确认服务端发生了内存泄漏。

### 18.1.1 客户端大量超时

---

查看 SLB 入口流量，发现客户端最初发生超时，并没有明显的流量变化，排除突





发流量高峰导致系统过载。分析服务端日志发现，在客户端超时时间段，后端的缓存服务出现了问题，缓存查询耗时比客户端的超时时间还长，因此导致了客户端超时。

由于采用的是 HTTP/1.1，客户端超时之后，需要关闭当前链路，重新发起 HTTPS 连接，因此日志中会大量打印读取响应超时和客户端关闭连接的异常。

### 18.1.2 服务端内存泄漏原因分析

分析清楚故障场景之后，模拟故障进行测试，使用 Dump 查看服务端内存文件，分析泄漏点，如图 18-1 所示。

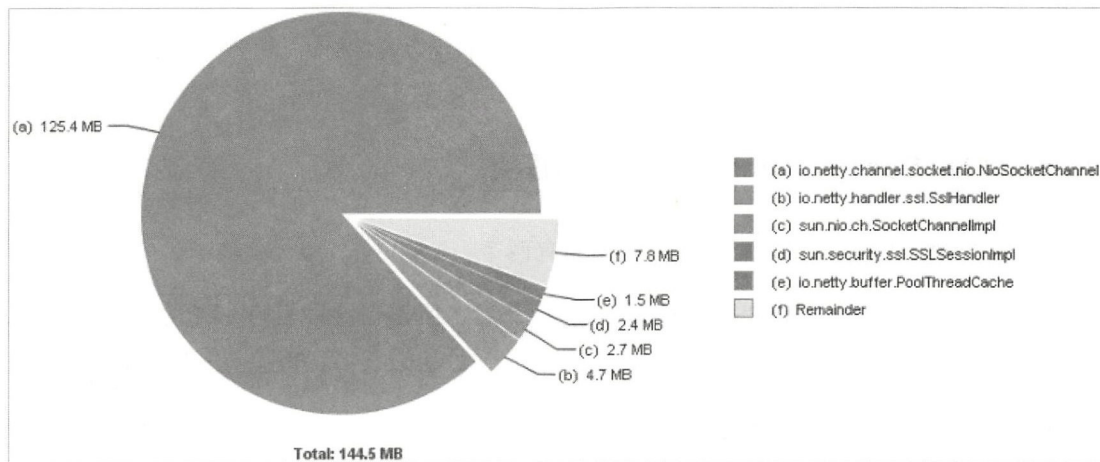


图 18-1 内存占用 Top 图

从对象内存占用排行可以看出，Netty 的 NioSocketChannel 内存占用排名第一，通过表格查看 NioSocketChannel 对象的内存占用详情，如图 18-2 所示。

共有 5729 个 NioSocketChannel 对象，占用 131MB 左右的内存。通过内存占用情况分析，可以确认导致服务端内存泄漏的是 NioSocketChannel。



## Netty 进阶之路：跟着案例学 Netty

▼ Biggest Top-Level Dominator Packages			
Package	Retained Heap	Retained Heap, %	# Top Dominators
<all>	151,487,640	100.00%	48,801
io	139,270,696	91.94%	8,777
netty	139,270,696	91.94%	8,777
channel	132,298,536	87.33%	7,425
socket	131,485,392	86.80%	6,016
nio	131,485,392	86.80%	6,014
NioSocketChannel	131,455,464	86.78%	5,729
handler	5,001,992	3.30%	529
ssl	5,001,880	3.30%	516
SslHandler	4,966,736	3.28%	249
buffer	1,773,016	1.17%	207
PoolThreadCache	1,578,976	1.04%	9
Σ Total: 3 entries	139,073,544		8,161

图 18-2 NioSocketChannel 内存占用详情

### 18.1.3 NioSocketChannel 泄漏原因探究

导致 NioSocketChannel 泄漏的可能原因有两个。

(1) 代码有缺陷，HTTPS 客户端关闭连接之后，服务端没有正确关闭连接。

(2) 服务端负载比较重，客户端超时之后的断连和重连速度超过服务端关闭连接速度，导致服务端的 NioSocketChannel 发生积压。随着积压数的增加，导致占用的内存快速增加，频繁 GC 使得服务端处理更慢，积压更严重，最终导致 OOM 异常。

明确方向之后，具体的定位策略有 3 点。

(1) 通过 netstat 命令查看服务端的端口连接状态，是不断地创建和回收连接，还是服务端没有关闭连接导致连接一直增长。

(2) 查看 NioSocketChannel 的状态，是全部没关闭，还是部分关闭、部分打开。

(3) 停止压测一段时间，观察连接数，以及服务端的内存占用情况，看服务端是否可以自动恢复。

在压测过程中，动态采集 HTTPS 的连接状态，发现超时的连接被服务端关闭，如图 18-3 所示。



```
C:\SE\netty>netstat -an|findstr 8443|find "ESTABLISHED" /c
11434

C:\SE\netty>netstat -an|findstr 8443|find "CLOSE_WAIT" /c
0

C:\SE\netty>netstat -an|findstr 8443|find "TIME_WAIT" /c
594
```

图 18-3 HTTPS 连接状态数据采集

接着通过 OQL 在内存堆栈中查询 NioSocketChannel 的连接状态，其中处于关闭状态的连接数为 25，如图 18-4 所示。

```
select * from io.netty.channel.socket.nio.NioSocketChannel s where s.ch.isOutputOpen = false
```

Class Name	Shallow Heap	Retained Heap
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc880c0d8	104	1,344
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc880abb8	104	1,344
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc880a288	104	1,344
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc8533f28	104	1,344
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc847bc58	104	1,344
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc8465650	104	1,344
> io.netty.channel.socket.nio.NioSocketChannel @ 0xc8465178	104	1,344
Σ Total: 19 of 25 entries; 6 more		

图 18-4 处于关闭状态的 NioSocketChannel 连接查询结果

服务端尚未主动关闭的 NioSocketChannel 实例个数为 5806，如图 18-5 所示。

```
select * from io.netty.channel.socket.nio.NioSocketChannel s where s.ch.isOutputOpen = true
```

Class Name	Shallow Heap	Retained Heap
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec8c890	104	24,000
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec7c710	104	24,000
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec76330	104	24,000
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec6e708	104	24,000
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec5ec08	104	24,000
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec53398	104	24,000
> io.netty.channel.socket.nio.NioSocketChannel @ 0xfec4de60	104	24,000
Σ Total: 8 of 5,806 entries; 5,798 more		

图 18-5 服务端尚未关闭的 NioSocketChannel 实例查询结果

从 OQL 查询可以看出，内存中尚有被服务端关闭但是还没来得及被 GC 回收的 NioSocketChannel 对象，证明客户端超时关闭连接后，服务端感知了连接关闭事件并主动关闭了连接。

当客户端发生大量超时、服务端占用内存飙升时，停止压测，观察服务端是否可以自



动恢复。停止压测观察服务端内存使用情况，如图 18-6 所示。

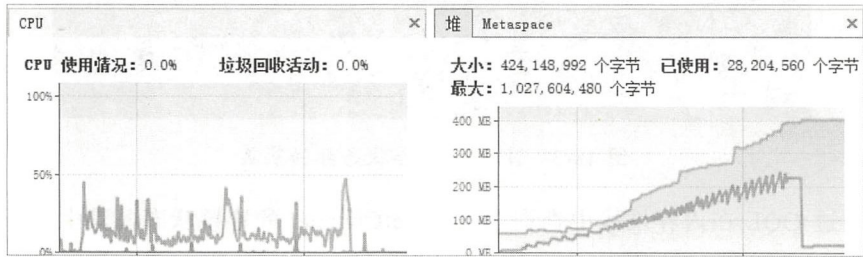


图 18-6 停止压测观察服务端内存使用情况

可以看出，停止压测一段时间之后，服务端的堆内存正常回收，说明并不存在由于忘记释放 NioSocketChannel 导致的内存泄漏问题。

同步观察 HTTPS 的连接数，发现停止压测一段时间之后，连接数逐步下降，最终降为 0，如图 18-7 所示（一段时间没消息，服务端会主动关闭连接）。

```
C:\SE\netty>netstat -an|findstr 8443
TCP    0.0.0.0:8443          0.0.0.0:0           LISTENING
TCP    127.0.0.1:8443        127.0.0.1:54067      TIME_WAIT
TCP    127.0.0.1:8443        127.0.0.1:54079      TIME_WAIT
TCP    [::]:8443             [::]:0               LISTENING

C:\SE\netty>netstat -an|findstr 8443
TCP    0.0.0.0:8443          0.0.0.0:0           LISTENING
TCP    [::]:8443             [::]:0               LISTENING
```

图 18-7 停止压测观察 HTTPS 连接数

经过上述分析得知，并不是由于服务端忘记关闭 NioSocketChannel 导致内存泄漏的，而是由于服务端关闭 NioSocketChannel 的速度没有客户端接入速度快导致 NioSocketChannel 缓慢积压，当积压到一定数量，无法在新生代被 GC，所以达到晋升阈值后被复制到老年代，引起老年代 GC，最终导致 OOM 异常。

#### 18.1.4 高并发场景下缺失的可靠性保护

由于涉及客户端和服务端的 RPC 调用，因此对业务组网进行分析，如图 18-8 所示。

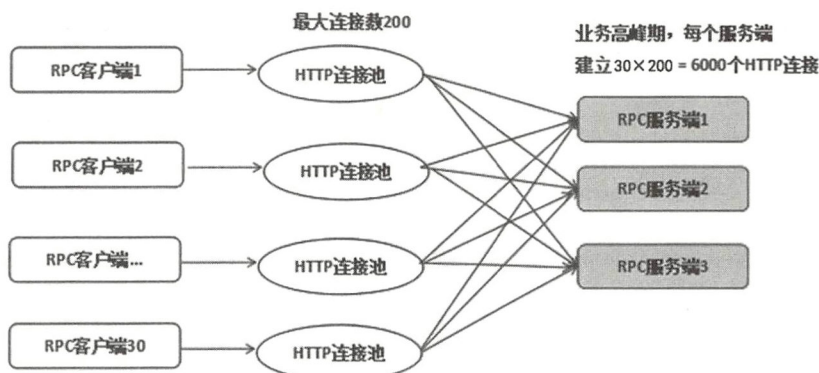


图 18-8 业务组网图

客户端采用 HTTP 连接池的方式与服务端进行 RPC 调用，单个客户端连接池上限为 200，客户端部署了 30 个实例，而服务端只部署了 3 个实例。在业务高峰期，每个服务端需要处理 6000 个 HTTP 连接，服务端时延增大之后，导致客户端批量超时，超时之后客户端会关闭连接重新发起 connect 操作，在某个瞬间，几千个 HTTPS 连接同时发起 SSL 握手操作，由于服务端此时也处于高负荷运行状态，导致部分连接 SSL 握手失败或者超时，超时之后客户端继续重连，进一步加重服务端的处理压力，最终导致服务端来不及释放客户端关闭的连接，引起 NioSocketChannel 大量积压，最终导致 OOM 异常。

从客户端的运行日志可以看到一些 SSL 握手发生了超时，如图 18-9 所示。

```

javax.net.ssl.SSLException: handshake timed out
    at io.netty.handler.ssl.SslHandler.handshake(...) (Unknown Source)
javax.net.ssl.SSLException: handshake timed out
    at io.netty.handler.ssl.SslHandler.handshake(...) (Unknown Source)
javax.net.ssl.SSLException: handshake timed out
    at io.netty.handler.ssl.SslHandler.handshake(...) (Unknown Source)
javax.net.ssl.SSLException: handshake timed out
    at io.netty.handler.ssl.SslHandler.handshake(...) (Unknown Source)
io.netty.channel.ConnectTimeoutException: connection timed out: /127.0.0.1:8443
    at io.netty.channel.nio.AbstractNioChannel$AbstractNioUnsafe$1.run(AbstractNioChannel.java:267)
    at io.netty.util.concurrent.PromiseTask$RunnableAdapter.call(PromiseTask.java:38)
    at io.netty.util.concurrent.ScheduledFutureTask.run(ScheduledFutureTask.java:127)
    at io.netty.util.concurrent.AbstractEventExecutor.safeExecute(AbstractEventExecutor.java:163)
    at io.netty.util.concurrent.SingleThreadEventExecutor.runAllTasks(SingleThreadEventExecutor.java:404)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:464)
  
```

图 18-9 SSL 握手超时日志

通过对服务端业务高峰期线程堆栈的分析发现，系统忙于处理 SSL 握手，处理客户端 HTTPS 连接接入是热点，如图 18-10 所示。

```
java.lang.Thread.State: BLOCKED (on object monitor)
    at javax.crypto.JceSecurity.getVerificationResult(JceSecurity.java:171)
    - waiting to lock <0x00000000eccc4308> (a java.lang.Class for javax.crypto.JceSecurity)
    at javax.crypto.JceSecurity.canUseProvider(JceSecurity.java:199)
    at javax.crypto.KeyGenerator.nextSpi(KeyGenerator.java:340)
    - locked <0x00000000ec133618> (a java.lang.Object)
    at javax.crypto.KeyGenerator.<init>(KeyGenerator.java:168)
    at javax.crypto.KeyGenerator.getInstance(KeyGenerator.java:223)
    at sun.security.ssl.JsseJce.getKeyGenerator(JsseJce.java:251)
    at sun.security.ssl.Handshaker.calculateMasterSecret(Handshaker.java:1209)
    at sun.security.ssl.Handshaker.calculateKeys(Handshaker.java:1157)
    at sun.security.ssl.ServerHandshaker.processMessage(ServerHandshaker.java:292)
    at sun.security.ssl.Handshaker.processLoop(Handshaker.java:1026)
    at sun.security.ssl.Handshaker$1.run(Handshaker.java:966)
    at sun.security.ssl.Handshaker$1.run(Handshaker.java:963)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.security.ssl.Handshaker$DelegatedTask.run(Handshaker.java:1416)
    - locked <0x00000000ec007d08> (a sun.security.ssl.SSLEngineImpl)
    at io.netty.handler.ssl.SslHandler.runDelegatedTasks(SslHandler.java:1435)
```

图 18-10 服务端 SSL 相关线程堆栈

服务端并没有对客户端的连接数做限制，导致尽管 ESTABLISHED 状态的连接数并不会超过 6000 这个上限，但是由于一些 SSL 连接握手失败，再加上积压在服务端的连接并没有及时被释放，最终引起了 NioSocketChannel 的大量积压。

客户端也没有流控机制，只要连接数不够用，就会一直创建连接，达到连接池配置的最大连接数。正是由于客户端和服务端都没有对高并发时大量的 HTTPS 链路断连和重连进行保护，导致了服务端 OOM 异常，业务中断。

## 18.2 功能层面的可靠性优化

问题的根本原因定位之后，可以从功能和架构层面做优化，由于架构优化改动较大，所以首先以快速解决问题为目标提升服务端的可靠性，通过较小的改动解决当前的业务问题。

## 18.2.1 Netty HTTPS 服务端可靠性优化

### 1. HTTPS 并发连接数流控

在服务端增加对客户端并发连接数的控制，服务端 HTTPS 连接数流控原理如图 18-11 所示。

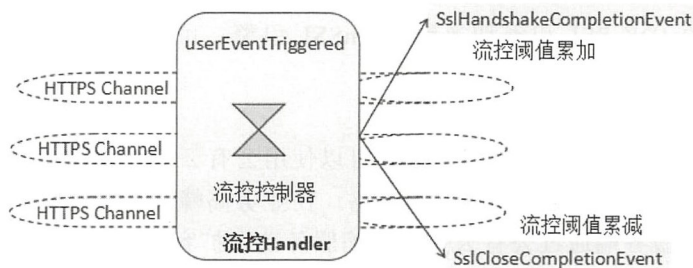


图 18-11 服务端 HTTPS 连接数流控原理

基于 Netty 的 Pipeline 机制，可以对 SSL 握手成功、SSL 连接关闭做切面拦截（类似于 Spring 的 AOP 机制，但是没采用反射机制，性能更高），通过流控切面接口，对 HTTPS 连接进行计数，根据计数器进行流控，服务端的流控算法如下。

（1）获取流控阈值。

（2）从全局上下文中获取当前的并发连接数，与流控阈值对比，如果小于流控阈值，则对当前的计数器进行原子自增，允许客户端连接。

（3）如果等于或者大于流控阈值，则抛出流控异常给客户端。

（4）SSL 连接关闭时，获取上下文中的并发连接数，进行原子自减。

在实现服务端流控时，需要注意如下几点。

（1）流控的 ChannelHandler 声明为 @ChannelHandler.Sharable，这样创建一个全局流控实例，就可以在所有的 SSL 连接中共享。

（2）通过 userEventTriggered 方法拦截 SslHandshakeCompletionEvent 和 SslCloseCompletionEvent 事件，在 SSL 握手成功和 SSL 连接关闭时更新流控计数器。

（3）流控并不是仅针对 ESTABLISHED 状态的 HTTP 连接，而是针对所有状态的连接，



因为客户端关闭连接，并不意味着服务端也同时关闭连接，只有触发 `SslCloseCompletion-Event` 事件时，服务端才真正关闭了 `NioSocketChannel`，GC 才会回收连接关联的内存。

(4) 流控 `ChannelHandler` 会被多个 `NioEventLoop` 线程调用，因此对于相关的计数器更新等操作，要保证并发安全性，避免使用全局锁，可以通过原子类等提升性能。

## 2. 切换 JDK SSL 引擎到 OpenSSL

将系统默认的 JDK SSL 引擎切换到 OpenSSL 引擎，可提升 SSL 的握手和通信性能。

## 3. HTTPS 服务端弹性伸缩

由于 HTTPS 服务端是无状态的，因此可以使用公有云的弹性伸缩服务，针对服务端的 CPU 和内存资源占用，配置弹性伸缩策略，在业务高峰期通过动态扩容的方式，保障业务的平稳运行。服务端通过公有云弹性伸缩服务动态扩容如图 18-12 所示。

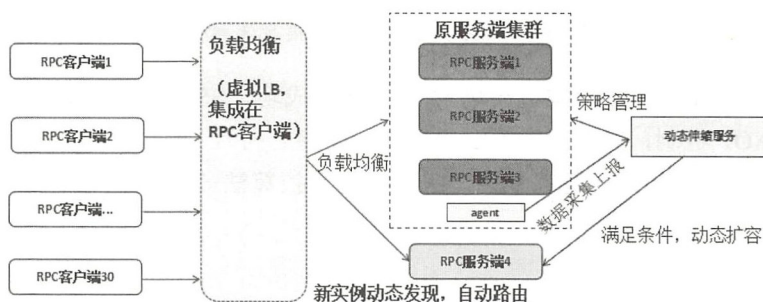


图 18-12 服务端通过公有云弹性伸缩服务动态扩容

## 18.2.2 HTTPS 客户端优化

### 1. HTTPS 连接池调整

由于生产环境客户端和服务端部署实例数是 10:1 的关系，服务端采用 Netty 构建，它的 I/O 通信通常不是瓶颈，如果服务端返回响应慢，说明服务端处理负荷已经比较重，或者服务端依赖的缓存、数据库、第三方服务等处理比较慢，拖累了服务端。此时，如果客户端一味地增加 HTTPS 连接数，并不能改善服务端的处理性能，反而会加剧它的负担，导致服务端响应更慢。

鉴于此，将客户端的连接池上限从 200 调整为 50，降低业务大批量超时重建 HTTPS



连接对服务端的冲击。

## 2. 超时时间调整

业务根据实验室的性能测试数据设置了响应超时时间，由于实验室数据比较好，因此把超时时间设置得很短。但是通过对生产环境的性能 KPI 数据采集分析发现，实际业务的平均处理时间比原来评估的要长一些，过短的超时时间导致了一定比例的超时失败，尽管失败率在可控范围内，但是由于 HTTP 超时之后会关闭连接，导致部分连接从长连接变成了短连接，而且重建 SSL 链路本身也比较耗时，这会影响系统的性能。

根据性能 KPI 数据分析，适当地调长了客户端超时时间，由于调长超时时间之后可能导致客户端线程同步阻塞时间变长，因此也同步调大了 Tomcat 工作线程的大小，以减小客户端发生同步阻塞时对系统吞吐量的影响。

## 18.3 架构层面的可靠性优化

功能优化只能缓解系统当前遇到的问题，并不能从根本上解决问题。通过端到端的架构优化，可以更优雅和合理地解决业务面临的性能和可靠性挑战。

### 18.3.1 端到端架构问题剖析

当前架构的业务调用模型如图 18-13 所示。

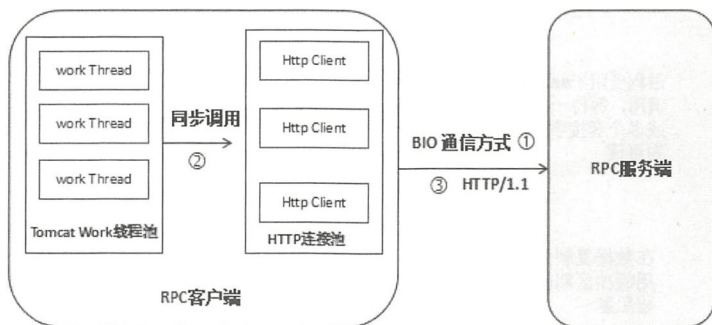


图 18-13 当前架构的业务调用模型

当前架构主要存在如下问题。

- (1) 客户端采用同步阻塞式 HTTP 调用，I/O 效率不高，调用线程容易被阻塞。
- (2) 客户端采用同步阻塞的方式进行 RPC 调用，如果服务端响应慢，容易阻塞调用方的线程。
- (3) 客户端和服务端采用 HTTP/1.1，由于 HTTP/1.1 是无状态的“一请求一应答”工作模式，所以单个链路通信效率低，需要创建大量链路来提升 I/O 性能。

### 18.3.2 HTTP Client 切换到 NIO

采用同步 HTTP Client 的主要问题如下。

- (1) 由于 I/O 读写是阻塞的，所以调用线程很容易被网络 I/O 阻塞。
- (2) 无法充分利用硬件资源，当调用线程被 I/O 阻塞时，CPU 资源闲置，但是系统的吞吐量却上不来。

切换到 Netty 的异步 HTTP Client 后，利用 I/O 多路复用技术，把多个 I/O 的阻塞复用到同一个 select 方法的阻塞上，使得系统在单线程的情况下可以同时处理多个客户端连接。与传统的多线程/多进程模型相比，I/O 多路复用的最大优势是系统开销小，系统不需要创建新的进程或者线程，也不需要维护这些进程和线程的运行，减少了系统的维护工作量，节省了系统资源，I/O 多路复用技术工作原理如图 18-14 所示。

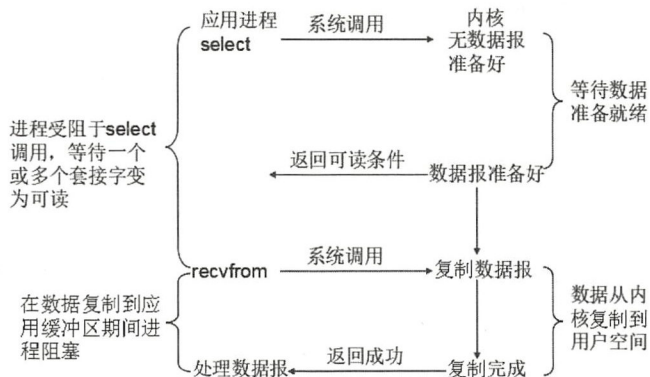


图 18-14 I/O 多路复用技术工作原理

相关代码示例如下：

---

```
//代码省略
EventLoopGroup group = new NioEventLoopGroup(1);

try {
    Bootstrap b = new Bootstrap();
    b.group(group)
      .channel(NioSocketChannel.class)
      .handler(new ChannelInitializer<SocketChannel>() {
          @Override
          protected void initChannel(SocketChannel ch) throws
Exception {

              ChannelPipeline p = ch.pipeline();
              if (sslCtx != null) {
                  p.addLast(sslCtx.newHandler(ch.alloc()));
              }
              p.addLast(new HttpClientCodec());
              p.addLast(new HttpObjectAggregator(10 * 1024 * 1024));
          }
      });

    for(int i = 0 ; i < corePoolSize; i++)
        b.connect(host, port).sync().channel();

//代码省略
}
```

---

### 18.3.3 同步 RPC 调用切换到异步调用

#### 1. NIO 与 RPC 调用方式的关系

实际上，通信框架基于 NIO 实现，并不意味着 RPC 框架就支持异步服务调用，两者本质上不是同一个层面的事情。NIO 只解决了通信层面的异步问题，跟服务调用的异步没有必然关系，也就是说，即便采用传统的 BIO 通信，依然可以实现异步服务调用，只不过通信效率和可靠性比较差而已。

下面对异步服务调用和通信框架的关系进行说明，如图 18-15 所示。

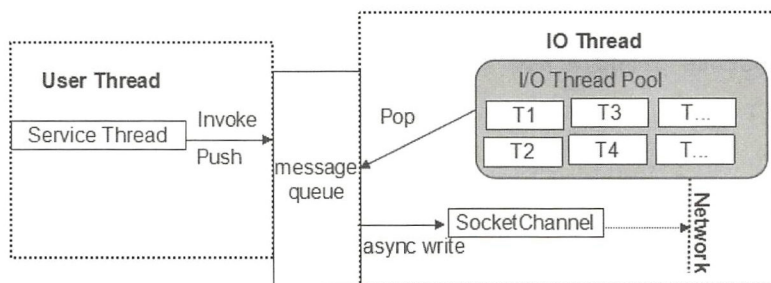


图 18-15 异步服务调用和通信框架的关系

用户发起远程服务调用之后，经历层层业务逻辑处理、消息编码，最终序列化的消息会被放入通信框架的消息队列。业务线程可以选择同步等待，也可以选择直接返回，通过消息队列的方式实现业务层和通信层的分离是比较成熟、典型的做法，现代的 RPC 框架或者 Web 服务器很少直接使用业务线程进行网络读写。

从图 18-15 可以看出，采用 NIO 还是 BIO 对上层的业务是不可见的，双方的汇聚点就是消息队列，在 Java 实现中它通常就是一个 Queue。业务线程将消息放入发送队列，可以选择主动等待或者立即返回，跟通信框架是否采用 NIO 没有任何关系。

## 2. 同步服务调用的缺点

同步服务调用是最常用的一种服务调用方式，它的工作原理和使用都非常简单，RPC 框架默认都需要支持这种调用方式。

它的工作原理：客户端发起远程服务调用请求，用户线程完成消息序列化之后，将消息投递到通信框架，然后同步阻塞，等待通信线程发送请求并接收应答，唤醒同步等待的用户线程，用户线程获取应答后返回。同步服务调用流程如图 18-16 所示。

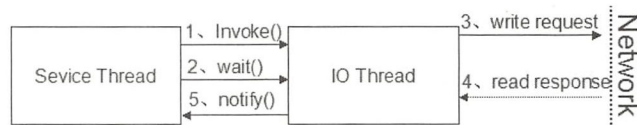


图 18-16 同步服务调用流程

同步服务调用最大的缺点就是，一旦服务端响应慢，RPC 客户端线程就会同步阻塞等待响应，只要某个接口的消息慢，就会导致后续所有接口消息都在线程池队列中排队，整

个系统的可靠性和性能都较差。

### 3. 异步服务调用设计

Java 8 的 `CompletableFuture` 提供了非常丰富的异步功能，它可以帮助用户简化异步编程的复杂性，通过 Lambda 表达式方便地编写异步回调逻辑，除了普通的异步回调接口，它还提供了多个异步操作结果转换，以及与、或等条件表达式的编排能力，方便对多个异步操作结果进行逻辑编排。

`CompletableFuture` 提供了大约 20 类比较实用的异步 API，如图 18-17 所示。

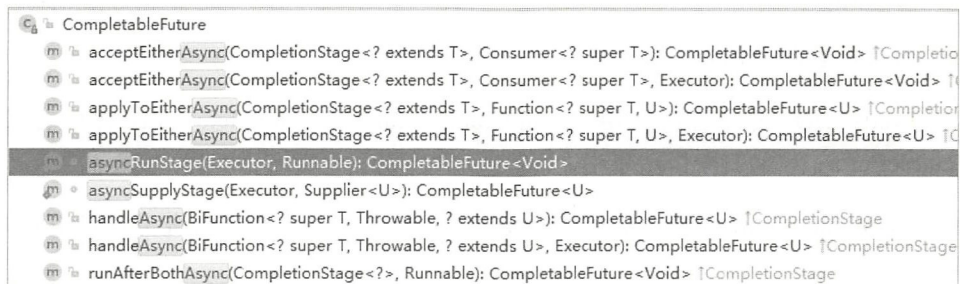


图 18-17 CompletableFuture 异步 API 定义

利用 JDK 的 `CompletableFuture` 与 Netty 的 NIO，可以非常方便地实现异步 RPC 调用，异步 RPC 调用设计原理如图 18-18 所示。

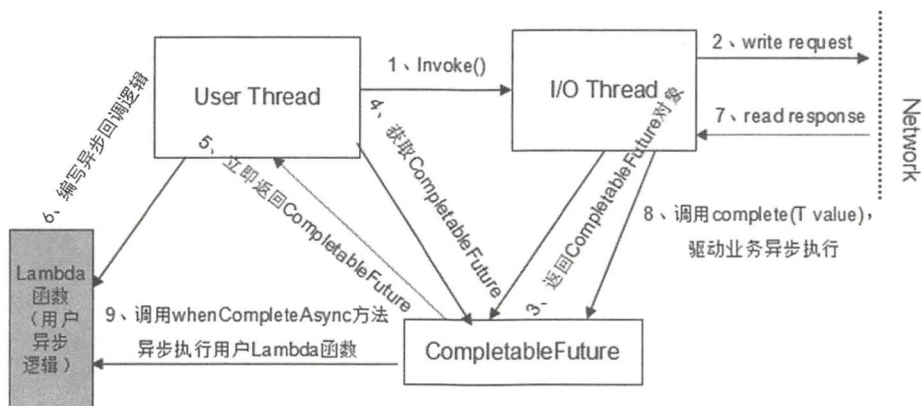


图 18-18 异步 RPC 调用设计原理

异步服务调用的工作流程如下。



- (1) 消费者通过 RPC 框架调用服务端。
- (2) Netty 异步发送 HTTP 请求消息，如果没有发生 I/O 异常就正常返回。
- (3) HTTP 请求消息发送成功后，I/O 线程构造 `CompletableFuture` 对象，设置到 RPC 上下文中。
- (4) 用户线程通过 RPC 上下文获取 `CompletableFuture` 对象。
- (5) 不阻塞用户线程，立即返回 `CompletableFuture` 对象。
- (6) 通过 `CompletableFuture` 编写 Function 函数，在 Lambda 函数中实现异步回调逻辑。
- (7) 服务端返回 HTTP 响应，Netty 负责反序列化工作。
- (8) Netty I/O 线程通过调用 `CompletableFuture` 的 `complete` 方法将应答设置到 `CompletableFuture` 对象的操作结果中。
- (9) `CompletableFuture` 通过 `whenCompleteAsync` 等方法异步执行业务回调逻辑，实现 RPC 调用的异步化。

#### 4. 异步服务调用的优势

异步服务调用相比同步调用有两个优点。

- (1) 化串行为并行，提升服务调用效率，减少业务线程阻塞时间。
- (2) 化同步为异步，避免业务线程阻塞。

假如一次应用市场首页访问需要调用多个服务接口，采用同步调用方式，如图 18-19 所示。

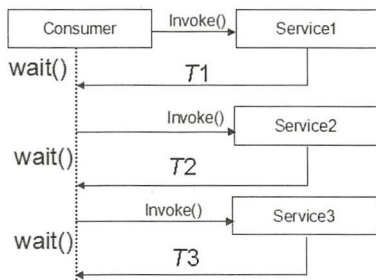


图 18-19 同步多服务调用

由于每次服务调用都是同步阻塞的，三个服务调用总耗时为  $T = T_1 + T_2 + T_3$ 。下面看采用异步服务调用的效果，如图 18-20 所示。

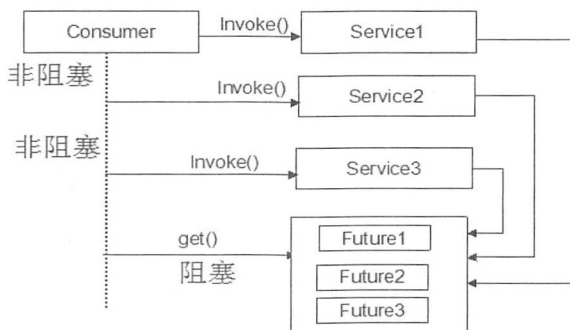


图 18-20 异步多服务调用

采用异步服务调用模式，最后调用三个服务异步操作结果 Future 的 get 方法同步等待应答，它的总执行时间  $T = \text{Max}(T_1, T_2, T_3)$ ，相比同步服务调用，性能提升非常明显。

### 18.3.4 协议升级到 HTTP/2

#### 1. HTTP/1.1 存在的问题

如果 HTTP 栈采用了异步非阻塞 I/O 模型（例如 Netty），则可以解决同步阻塞 I/O 的问题，带来如下收益。

（1）同一个 I/O 线程可以并行处理多个客户端连接，有效减少了 I/O 线程数量，提升了资源的利用率。

（2）读写等 I/O 操作都是非阻塞的，不会因为服务端响应慢、网络时延长等导致 I/O 线程被阻塞。

相比采用私有协议构建的 RPC 框架，例如 MessagePack、Thrift 等，采用了非阻塞 I/O 的 HTTP/1.1 仍然存在性能问题，如图 18-21 所示。

由于 HTTP 是无状态的，客户端发送请求之后，必须等到接收服务端响应之后，才能继续发送下一个请求消息。在某一个时刻，链路上只存在单向的消息流，实际上把 TCP 的全双工模式变成了单工模式。

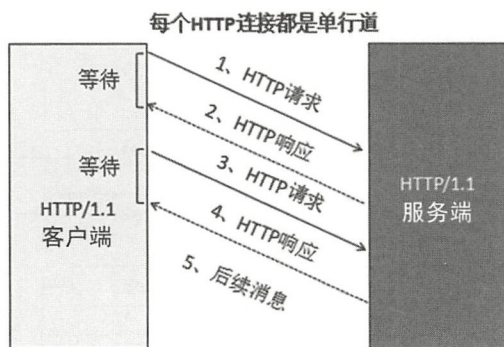


图 18-21 HTTP/1.1 性能问题

如果服务端响应耗时较大，则单个 HTTP 连接的通信性能严重下降，只能通过不断地新建连接来提升 I/O 性能。但这也会带来很多副作用，例如句柄数的增加、I/O 线程的负担加重等。除了无状态导致的链路传输性能差，HTTP/1.1 还存在如下几个影响性能的问题。

(1) 超时关闭连接问题：HTTP 客户端超时之后，由于协议是无状态的，客户端无法对请求和响应进行关联，只能关闭连接进行重连，反复地断连和重连会增加成本和时延（如果客户端选择不关闭连接，继续发送新的请求，服务端可能会把上一条客户端认为超时的响应返回，也可能按照 HTTP 规范直接关闭连接，无论哪种处理方式，都会导致连接被关闭）。如果采用传统的 RPC 私有协议，请求和响应可以通过消息 ID 或者 SessionId 关联，某条消息的超时并不需要关闭连接，只需要丢弃超时消息即可。

(2) HTTP 本身包含文本类型的协议消息头，占用一些字节。另外，采用 JSON 等文本类序列化方式，报文相比传统的私有 RPC 协议也大很多，增加了网络传输开销。

(3) 服务端无法主动推送响应。

## 2. HTTP/2 streaming 调用

如果选择 Restful API 或者 HTTP 作为内部 RPC 接口协议，则建议使用 HTTP/2 来承载，它的优点包括支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等。可以有效地解决传统 HTTP/1.1 遇到的问题，效果与 TCP 私有协议栈相近。

基于 HTTP/2 的 RPC 调用，可以参考和借鉴 gRPC Java 版的实现策略。基于 HTTP/2，gRPC 提供了三种 streaming 模式。

(1) 服务端 streaming。

(2) 客户端 streaming。

(3) 服务端和客户端双向 streaming。

服务端 streaming 模式指客户端发送 1 个 RPC 请求，服务端返回  $N$  个响应，响应可以单独返回，也可以一起返回，如图 18-22 所示。

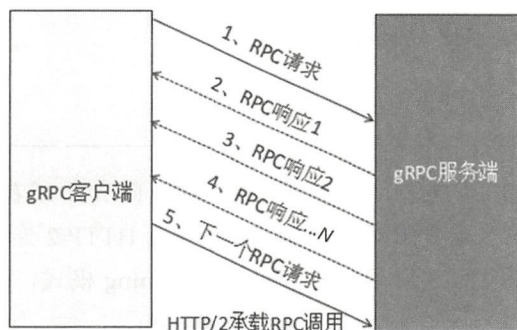


图 18-22 gRPC 服务端 streaming

服务端 streaming 模式适用的场景：客户端发送单个 RPC 请求，但是服务端可能返回的是一个结果列表，服务端不想等到所有的响应消息都组装完成才返回一个应答消息给客户端，而是处理完成一个就返回一个响应，直到服务端关闭 stream，通知客户端响应全部发送完成。

服务端 streaming 模式的本质：如果响应是列表，列表中的单个响应互相没关联关系，有些耗时长，有些耗时短，为了防止快的等慢的，可以处理完一个就返回一个响应，不需要等所有的都处理完统一返回响应。它可以压缩单个响应的时延，端到端提升用户的体验。

客户端 streaming 指的是客户端发送多个请求，服务端返回一个响应，多用于数据汇聚和汇总计算场景。

gRPC 客户端 streaming 的代码实现策略：异步服务调用获取请求 StreamObserver 对象，循环调用 requestObserver.onNext(point) 方法，异步发送请求消息到服务端。发送完成之后，调用 requestObserver.onCompleted()，通知服务端所有请求已经发送完成，可以接收服务端响应。代码示例如下（RouteGuideClient 类 RouteGuideClient 方法）：

//代码省略

```
StreamObserver<Point> requestObserver = asyncStub.recordRoute(responseObserver);
```

```

try {
    for (int i = 0; i < numPoints; ++i) {
        int index = random.nextInt(features.size());
        Point point = features.get(index).getLocation();
        info("Visiting point {0}, {1}", RouteGuideUtil.getLatitude(point),
            RouteGuideUtil.getLongitude(point));
        requestObserver.onNext(point);
    }
}
//代码省略

```

双向 streaming 指客户端发送  $N$  个请求，服务端返回  $N$  个或者  $M$  个响应，通过该特性可以充分利用 HTTP/2 的多路复用功能，在某个时刻，HTTP/2 连接上既有请求也有响应，实现了全双工通信，如图 18-23 所示。对于双向 streaming 模式，gRPC 只支持异步调用一种方式。

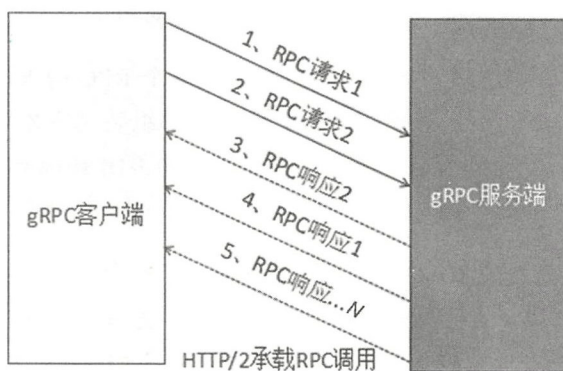


图 18-23 gRPC 双向 streaming

gRPC 服务调用支持同步和异步方式，同时也支持普通的 RPC 和 streaming 模式，可以满足不同的业务场景需求。对于 streaming 模式，可以充分利用 HTTP/2 的多路复用能力，实现在同一条 HTTP 连接上并行双向传输数据，有效地解决了 HTTP/1.1 的数据单向传输问题，在大幅减少 HTTP 连接的情况下，充分利用单个连接的性能，提高资源的使用率。

借鉴 gRPC 的设计思路，利用 HTTP/2 的多路复用，实现支持多种模式的 streaming 调用，可以极大地提升 RPC 调用的灵活性，以及提升系统的性能和可靠性。



## 18.4 总结

随着信息安全和个人隐私数据保护的加强，越来越多的 RPC 框架开始支持 SSL/TLS 传输通道加密，表面上增加数据加解密功能只对系统的性能有一些影响，但实际上系统的可靠性也会面临比较大的挑战，特别是高并发、低时延类的业务，一旦发生批量服务调用超时，就会导致大量的 SSL 链路重建，在业务高峰期，如果服务端可靠性设计欠缺，很有可能宕机，导致业务中断。

针对 HTTPS 服务的优化，除了功能上加强可靠性设计，更需要从架构层面做系统性的优化，借鉴 gRPC 的设计理念，基于 HTTP/2 构建异步流式 RPC 调用是个不错的选择。

## 第 19 章

---

# MQTT 服务接入超时案例

Netty 4.1 提供了 MQTT 协议栈，基于此可以非常方便地创建 MQTT 服务，尽管开发简单，但是在实际环境中会面临各种挑战，甚至会面临一些不遵循 MQTT 规范的端侧设备接入。

如果服务端没有考虑到各种异常场景，很难稳定运行，本章以生产环境 MQTT 服务无法提供接入服务为例，详细介绍 MQTT 服务和 Netty 在异常场景下的保护机制。

## 19.1 MQTT 服务接入超时问题

---

### 19.1.1 生产环境问题现象

---

生产环境的 MQTT 服务运行一段时间之后，发现新的端侧设备无法接入，连接超时。分析 MQTT 服务端日志，没有明显的异常，但是内存占用较高，查看连接数，发现有数十万个 TCP 连接处于 ESTABLISHED 状态，实际的 MQTT 连接数应该在 1 万个左右，显然这么多连接肯定存在问题。

由于 MQTT 服务端的内存是按照 2 万个左右连接数规模配置的，因此当连接数达到数十万个的规模之后，导致了服务端大量 SocketChannel 积压、内存暴涨、高频率 GC 和较长的 STW 时间，对端侧设备的接入造成了很大影响，部分设备 MQTT 握手超时，无法接入。

### 19.1.2 连接数膨胀原因分析

通过抓包分析发现，一些端侧设备并没有按照 MQTT 协议规范进行处理，包括：

(1) 客户端发起 CONNECT 连接，SSL 握手成功之后没有按照协议规范继续处理，例如发送 PING 命令。

(2) 客户端发起 TCP 连接，不做 SSL 握手，也不做后续处理，导致 TCP 连接被挂起。

由于服务端是严格按照 MQTT 规范实现的，上述端侧设备不按规范接入，实际上消息调度不到 MQTT 应用协议层。MQTT 服务端依赖 Keep Alive 机制进行超时检测，当一段时间接收不到客户端的心跳和业务消息时，就会触发心跳超时，关闭连接。针对上述两种接入场景，由于 MQTT 的连接流程没有完成，MQTT 协议栈不认为这个是合法的 MQTT 连接，因此心跳保护机制无法对上述 TCP 连接进行检测。客户端和服务端都没有主动关闭这个连接，导致 TCP 连接一直保持。

MQTT 连接建立过程如图 19-1 所示。

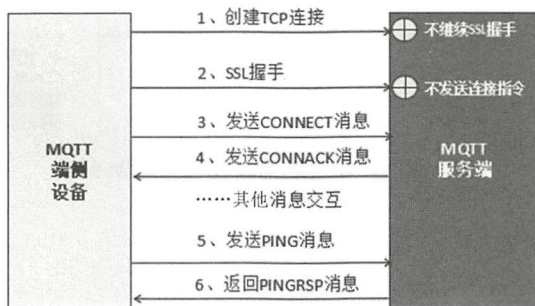


图 19-1 MQTT 连接建立过程

### 19.1.3 无效连接的关闭策略

针对这种不遵循 MQTT 规范的端侧设备，除了要求对方按照规范修改，服务端还需

要做可靠性保护，具体策略如下。

(1) 端侧设备的 TCP 连接接入后，启动一个链路检测定时器加入 Channel 对应的 NioEventLoop。

(2) 链路检测定时器一旦触发，就主动关闭 TCP 连接。

(3) TCP 连接完成 MQTT 协议层的 CONNECT 之后，删除之前创建的链路检测定时器。

MQTT 无效连接检测机制如图 19-2 所示。

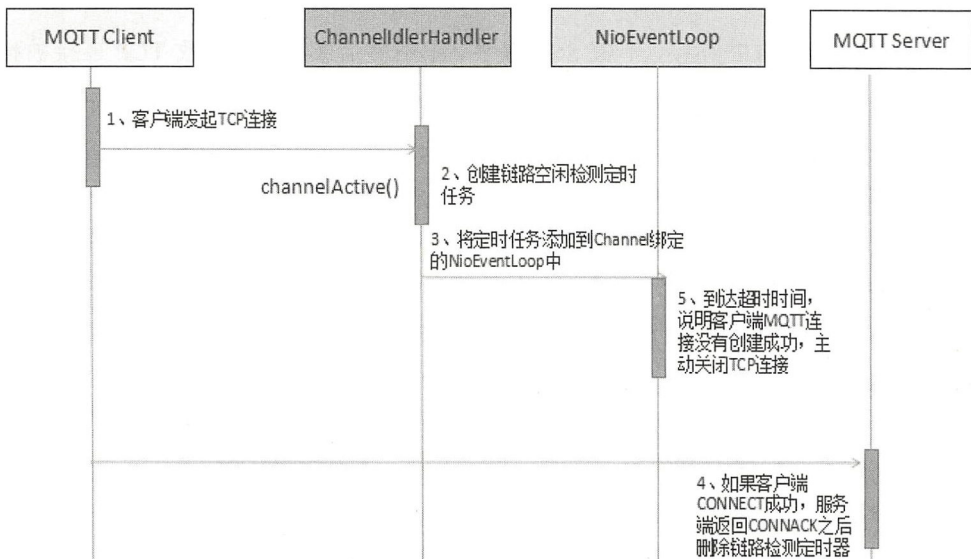


图 19-2 MQTT 无效连接检测机制

### 19.1.4 问题总结

生产环境升级版本之后，平稳运行，查看 MQTT 连接数，稳定在 1 万个左右，与预期一致，问题得到解决。

对于 MQTT 服务端，除了要遵循协议规范，还需要对那些不遵循规范的客户端接入进行保护，不能因为一些客户端没按照规范实现，导致服务端无法正常工作。系统的可靠

性设计更多的是在异常场景下保护系统的稳定运行。

## 19.2 基于 Netty 的可靠性设计

从应用场景看，Netty 是基础的通信框架，一旦出现问题，轻则需要重启应用，重则可能导致整个业务中断。它的可靠性会影响整个业务集群的数据通信和交换，在以分布式为主的软件架构体系中，通信中断就意味着整个业务中断，分布式架构对通信的可靠性要求非常高。

从运行环境看，Netty 会面临恶劣的网络环境，这就要求它自身的可靠性要足够好，平台能够解决的可靠性问题需要由 Netty 自身来解决，否则会导致上层用户关注过多的底层故障，降低 Netty 的易用性，同时增加用户的开发和运维成本。

Netty 的可靠性如此重要，它的任何故障都可能导致业务中断，产生巨大的经济损失。因此，Netty 在版本的迭代中不断加入新的可靠性特性来满足用户日益增长的高可靠和健壮性需求。

### 19.2.1 业务定制 I/O 异常

在大多数场景下，当底层网络发生故障的时候，应该由底层的 NIO 框架负责释放资源，处理异常等，上层的业务应用不需要关心底层的处理细节。但是，在一些特殊的场景下，用户可能需要关心这些异常，并针对这些异常进行定制处理，例如：

- (1) 客户端的断连和重连机制。
- (2) 消息的缓存重发。
- (3) 在接口日志中详细记录故障细节。
- (4) 运维相关功能，例如告警、触发邮件/短信等。

Netty 的处理策略是，发生 I/O 异常时，底层的资源由它负责释放，同时将异常堆栈信息以事件的形式通知给上层用户，由用户对异常进行定制。这种处理机制既保证了异常



处理的安全性，也向上层提供了灵活的定制能力。Netty 异常通知接口定义如图 19-3 所示。

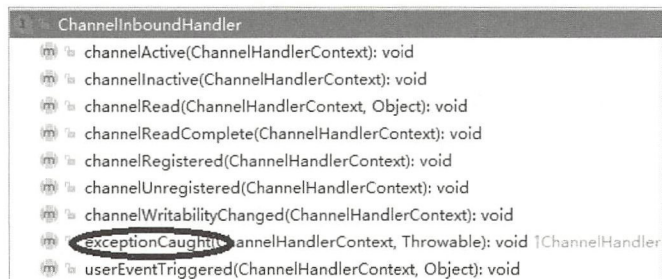


图 19-3 Netty 异常通知接口定义

### 19.2.2 链路的有效性检测

当网络发生单通、连接被防火墙挡住、长时间 GC 或者通信线程发生非预期异常时，链路会不可用且不易被及时发现。特别是如果异常发生在凌晨业务低谷期间，当早晨业务高峰到来时，由于链路不可用导致瞬间大批量业务失败或者超时，这将对系统的可靠性产生重大的威胁。

从技术层面看，要解决链路的可靠性问题，必须周期性地对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。

心跳检测机制分为三个层面。

(1) TCP 层面的心跳检测，即 TCP 的 Keep-Alive 机制，它的作用域是整个 TCP 协议栈。

(2) 协议层的心跳检测，主要存在于长连接协议中，例如 MQTT。

(3) 应用层的心跳检测，它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。心跳检测的目的就是确认当前链路是否可用，对方是否活着并且能够正常接收和发送消息。

作为高可靠的 NIO 框架，Netty 也提供了心跳检测机制，利用 IdleStateHandler 可以方便地实现业务层的心跳检测。

### 19.2.3 内存保护

NIO 通信的内存保护主要集中在如下几点。

(1) 链路总数的控制：每条链路都包含接收和发送缓冲区，链路个数太多容易导致内存溢出。

(2) 单个缓冲区的上限控制：防止非法长度或者消息过大导致内存溢出。

(3) 缓冲区内内存释放：防止因为缓冲区使用不当导致的内存泄漏。

(4) NIO 消息发送队列的长度上限控制。

#### 1. 防止内存池泄漏

为了提升内存的利用率，Netty 提供了内存池和对象池。但是，基于缓存池实现以后需要对内存的申请和释放进行严格的管理，否则很容易导致内存泄漏。

如果不采用内存池技术实现，每次对象都以方法的局部变量形式被创建，用完之后，只要不再继续引用它，JVM 会自动释放。但是，一旦引入内存池机制，对象的生命周期将由内存池负责管理，通常是全局引用（含线程级缓存），如果不显式释放，JVM 是不会回收这部分内存的。对于从内存池申请的对象，使用完毕一定要及时释放，防止内存泄漏。

#### 2. 缓冲区溢出保护

当我们对消息进行解码的时候，需要创建缓冲区（Netty 的 `ByteBuffer`）。缓冲区的创建方式通常有两种。

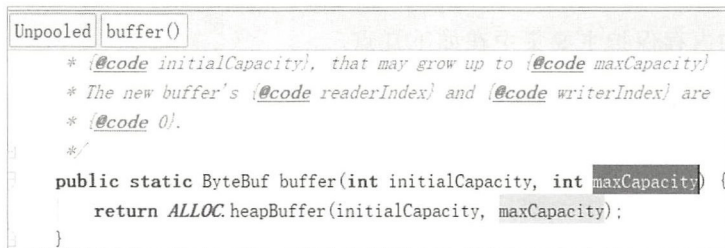
(1) 容量预分配，在实际读写过程中如果不够再扩展。

(2) 根据协议消息长度创建缓冲区。

在实际的商用环境中，如果遇到畸形码流攻击、协议消息编码异常、消息丢包等问题，可能会解析到一个超长的长度字段。我曾经遇到过类似问题，报文长度字段值竟然超过 2GB，由于代码的一个分支没有对长度上限进行有效保护，导致内存溢出。系统重启几秒后再次发生内存溢出，幸好及时定位到问题的根本原因，没有造成严重的事故。

Netty 提供了编解码框架，因此对于解码缓冲区的上限保护就显得非常重要，在实际项目中主要通过如下两种方式对缓冲区进行保护。

(1) 创建 ByteBuf 时对它的容量上限进行保护性设置，如图 19-4 所示。



```

Unpooled buffer()
    * {@code initialCapacity}, that may grow up to {@code maxCapacity}
    * The new buffer's {@code readerIndex} and {@code writerIndex} are
    * {@code 0}.
    */
    public static ByteBuf buffer(int initialCapacity, int maxCapacity) {
        return ALLOC.heapBuffer(initialCapacity, maxCapacity);
    }

```

图 19-4 ByteBuf 容量上限保护

(2) 在消息解码的时候，对消息长度进行判断，如果超过最大容量，则抛出解码异常，拒绝分配内存，以 LengthFieldBasedFrameDecoder 的 decode 方法为例进行说明，代码如下：

---

```

//代码省略
if (frameLength > maxFrameLength) {
    long discard = frameLength - in.readableBytes();
    tooLongFrameLength = frameLength;
    if (discard < 0) {
        in.skipBytes((int) frameLength);
    } else {
        discardingTooLongFrame = true;
        bytesToDiscard = discard;
        in.skipBytes(in.readableBytes());
    }
    failIfNecessary(true);
    return null;
}
//代码省略
}

```

---

### 3. 消息发送队列积压保护

Netty 的 NIO 消息发送队列 ChannelOutboundBuffer 并没有容量上限，它会随着消息的积压自动扩展，直到达到 0x7fffffff。

如果对方处理速度比较慢，会导致 TCP 滑窗长时间为 0；如果消息发送方发送速度过



快或者一次批量发送消息量过大，会导致 `ChannelOutboundBuffer` 的内存膨胀，可能会使系统的内存溢出。

建议业务配置合适的高水位（`writeBufferWaterMark`）对消息发送速度进行控制，同时在发送业务消息时，调用 `Channel` 的 `isWritable` 方法判断 `Channel` 是否可写，如果不可写则不要继续发送，否则会导致发送队列积压，出现 OOM 异常。

## 19.3 总结

可靠性设计的关键在于对非预期异常场景的保护，应用层协议栈会考虑应用协议异常时通信双方应该怎么正确处理异常，但是对于那些不遵循协议规范实现的客户端，协议规范是无法强制约束对方的，特别是在物联网应用中，面对各种厂家的不同终端设备接入，服务端需要应对各种异常。只有可靠性做得足够好，MQTT 服务才能更从容地应对海量设备的接入。



## 第 20 章

---

# Netty 实践总结

Netty 入门相对简单，但是要在实际项目中用好它，出了问题能够快速定位和解决，却并非易事。

本章对 Netty 的学习方式及问题定位技巧进行总结，以帮助大家更快地掌握 Netty，更好地为自己的业务服务。

## 20.1 Netty 学习策略

### 20.1.1 入门知识准备

#### 1. Java NIO 类库

需要熟悉和掌握的类库主要包括：

- (1) 缓冲区 Buffer。
- (2) 通道 Channel。







### （3）多路复用器 Selector。

首先介绍缓冲区（Buffer）的概念，Buffer 是一个对象，它包含一些要写入或者要读出的数据。在 NIO 类库中加入 Buffer 对象，体现了新库与原 I/O 的一个重要区别。在面向流的 I/O 中，可以将数据直接写入或者将数据直接读到 Stream 对象中。在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区的；在写入数据时，直接写入缓冲区。在任何时候访问 NIO 中的数据，都通过缓冲区进行操作。

缓冲区实质上是一个数组。通常它是一个字节数组（ByteBuffer），也可以使用其他种类的数组。但是一个缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问及维护读写位置（limit）等信息。

最常用的缓冲区是 ByteBuffer，一个 ByteBuffer 提供了一组功能用于操作 byte 数组。比较常用的就是 get 和 put 系列方法，如图 20-1 所示。

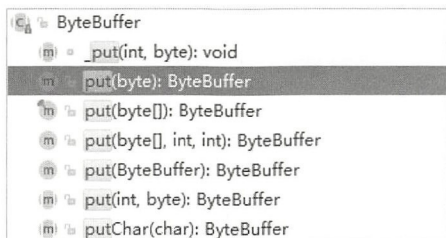


图 20-1 ByteBuffer 常用方法定义

Channel 是一个通道，可以通过它读取和写入数据，它就像自来水管一样，网络数据通过 Channel 读取和写入。通道与流的不同之处在于，通道是双向的，流只是在一个方向上移动（流必须是 InputStream 或者 OutputStream 的子类），而且通道可以用于读或写，或者同时用于读和写。因为 Channel 是全双工的，所以它可以比流更好地映射底层操作系统的 API。特别是在 UNIX 网络编程模型中，底层操作系统的通道都是全双工的，同时支持读和写操作。

比较常用的 Channel 是 SocketChannel 和 ServerSocketChannel，其中 SocketChannel 的继承关系如图 20-2 所示。

Selector 是 Java NIO 编程的基础，熟练地掌握 Selector 对于掌握 NIO 编程至关重要。多路复用器提供选择已经就绪的任务的能力。简单来讲，Selector 会不断地轮询注册在其上的 Channel，如果某个 Channel 上面有新的 TCP 连接接入、读和写事件，这个 Channel





就处于就绪状态，会被 Selector 轮询出来，然后通过 SelectionKey 获取就绪 Channel 的集合，进行后续的 I/O 操作。

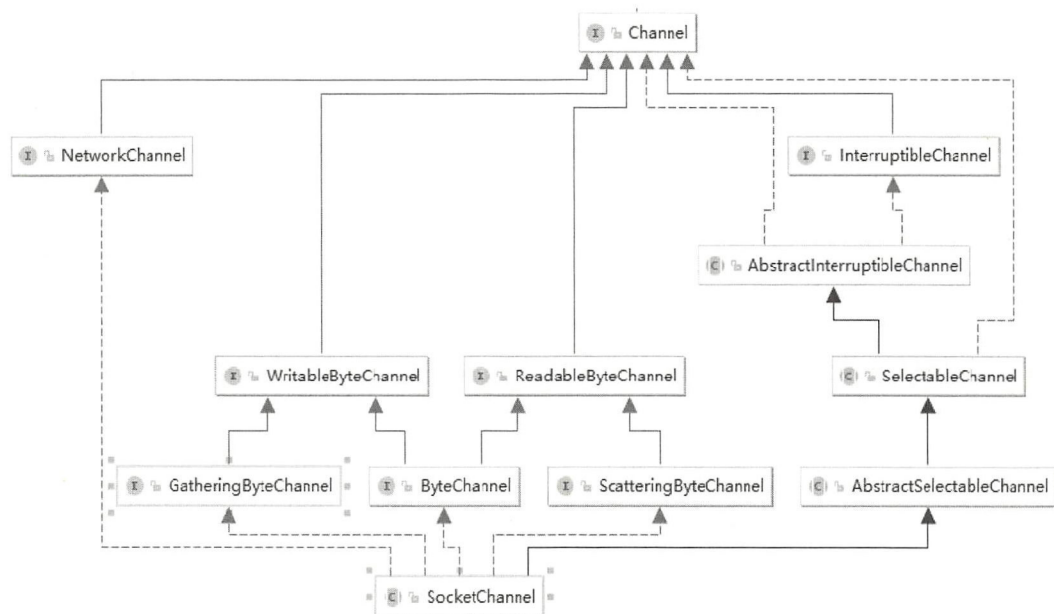


图 20-2 SocketChannel 的继承关系

## 2. Java 多线程编程

作为异步事件驱动、高性能的 NIO 框架，Netty 代码中大量运用了 Java 多线程编程技巧，熟练掌握多线程编程知识是掌握 Netty 的必备条件。

需要掌握的多线程编程相关知识包括：

- (1) Java 内存模型。
- (2) 关键字 synchronized。
- (3) 读写锁。
- (4) volatile 的正确使用。
- (5) CAS 指令和原子类。
- (6) JDK 线程池及各种默认实现。



## 20.1.2 Netty 入门学习

通过 Netty 官方文档，以及 Netty 自带的例子（example）来进行学习，可以先看自带的例子，然后动手编写和调试代码。把 echo 客户端连接创建、服务端连接接入、客户端消息发送和读取、服务端消息发送和读取流程及机制搞清楚，再学习其他的协议栈，例如 HTTP、MQTT 等。

## 20.1.3 项目实践

完成 Netty 基础知识的学习之后，就可以在项目中使用 Netty 了，只有通过实践才能真正掌握 Netty，如果项目暂时用不到 Netty，可以学习一些开源的 RPC 或者服务框架，看这些框架是怎么集成并使用 Netty 的，例如：

（1）gRPC，主要用到了 Netty 的 HTTP/2 协议栈。

（2）Vert.x，主要用到了 Netty 的 NIO 通信框架和 HTTP 协议栈，以及 EventLoop Reactive 线程模型。

建议通过“源码阅读 + 调试”的方式来学习，因为 Netty 的很多处理都是异步的，单纯地靠阅读代码很难真正掌握消息的处理流程。

## 20.1.4 Netty 源码阅读策略

当前 Netty 的源码规模已经非常庞大，全部阅读并掌握需要很长时间，建议读者根据自己的需要选择性地阅读。

首先需要掌握 Netty 的内核工作流程，主要包括客户端连接创建、客户端消息读写、服务端监听和客户端接入、服务端消息读写，涉及的核心类库如下。

（1）ByteBuf 和各种子类。

（2）Channel、Unsafe 及各种子类。

（3）ChannelPipeline 和 ChannelHandler。

（4）EventLoop 和 EventLoopGroup 及各种子类实现。



### （5）Future 和 Promise 及各种子类实现。

作为入门知识，Netty 客户端和服务端的创建流程需要熟练掌握，以服务端的创建流程为例，画出流程创建图，然后结合源码阅读，并进行调试，可以更高效地掌握相关知识，如图 20-3 所示。

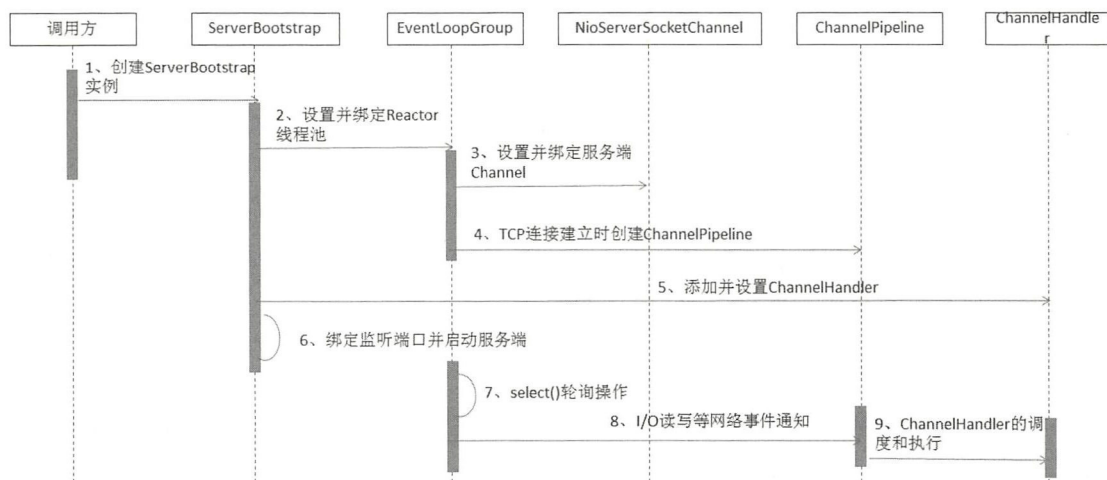


图 20-3 Netty 服务端创建流程

掌握了 Netty 内核工作原理之后，再结合业务的实际需求选择性地学习其他协议栈，例如 HTTP。

## 20.2 Netty 故障定位技巧

尽管 Netty 应用广泛，非常成熟，但是由于对 Netty 底层机制不太了解，用户在实际使用中还是会经常遇到各种问题，大部分问题都是业务使用不当导致的。Netty 使用者需要学习 Netty 的故障定位技巧，以便出了问题能够独立、快速地解决。

### 20.2.1 接收不到消息

如果业务的 ChannelHandler 接收不到消息，可能的原因如下。





(1) 业务的解码 ChannelHandler 存在缺陷, 导致消息解码失败, 没有投递到后端。

(2) 业务发送的是畸形或者错误码流 (例如长度错误), 导致业务解码 ChannelHandler 无法正确解码业务消息。

(3) 业务 ChannelHandler 执行了一些耗时或者阻塞操作, 导致 Netty 的 NioEventLoop 被挂住, 无法读取消息。

(4) 执行业务 ChannelHandler 的线程池队列积压, 导致新接收的消息排队, 没有得到及时处理。

(5) 对方确实没有发送消息。

定位策略如下。

(1) 在业务的首个 ChannelHandler 的 channelRead 方法中设置断点进行调试, 看是否能读取到消息。

(2) 在 ChannelHandler 中添加 LoggingHandler, 打印接口日志。

(3) 查看 NioEventLoop 线程的状态, 看是否发生了阻塞。

(4) 通过 tcpdump 抓包查看消息是否发送成功。

## 20.2.2 内存泄漏

通过 “jmap -dump:format=b,file=xx pid” 命令打印内存堆栈, 然后使用 MemoryAnalyzer 工具对内存占用情况进行分析, 查找内存泄漏点, 再结合代码进行分析, 定位内存泄漏的具体原因, 如图 20-4 所示。

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
> java.util.concurrent.ThreadPoolExecutor @ 0xe04c5bc0	72	155,649,872	96.29%
> com.sun.jmx.remote.internal.ArrayNotificationBuffer @ 0xe0119cb0	64	1,867,904	1.16%
> sun.misc.Launcher\$AppClassLoader @ 0xe0269a40	88	400,776	0.25%
> io.netty.buffer.PoolThreadCache @ 0xe04cb910	64	196,672	0.12%
> io.netty.buffer.PoolThreadCache @ 0xe00d9af8	64	195,648	0.12%
> io.netty.buffer.PoolThreadCache @ 0xe03c92a8	64	195,648	0.12%
> class sun.util.calendar.ZoneInfoFile @ 0xe01766a0 System Class	120	162,032	0.10%
> io.netty.channel.nio.NioEventLoopGroup @ 0xe0272a08	32	140,816	0.09%
> java.lang.Thread @ 0xe04c5a48 pool-3-thread-8 Thread	120	131,312	0.08%
> java.lang.Thread @ 0xe04c5c88 pool-3-thread-7 Thread	120	131,312	0.08%
> java.lang.Thread @ 0xe04c5e30 pool-3-thread-6 Thread	120	131,312	0.08%

图 20-4 通过 MemoryAnalyzer 工具分析内存堆栈







## 20.2.3 性能问题

如果出现性能问题，首先需要确认是 Netty 的问题还是业务的问题，通过 `jstack` 命令或者 `jvisualvm` 工具打印线程堆栈，按照线程 CPU 使用情况进行排序（`top -Hp` 命令采集），看线程在忙什么。通常如果采集几次都发现 Netty 的 NIO 线程堆栈停留在 `select` 操作上，说明 I/O 比较空闲，性能瓶颈不是 Netty，如图 20-5 所示。

```
at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
- locked <0x00000000eb9c4dc8> (a io.netty.channel.nio.SelectedSelectionKeySet)
- locked <0x00000000eb9841d0> (a java.util.Collections$UnmodifiableSet)
- locked <0x00000000eb982178> (a sun.nio.ch.WindowsSelectorImpl)
at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
at io.netty.channel.nio.SelectedSelectionKeySetSelector.select(SelectedSelectionKeySetSelector.java:62)
at io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:755)
at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:410)
at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:884)
at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
at java.lang.Thread.run(Thread.java:748)
```

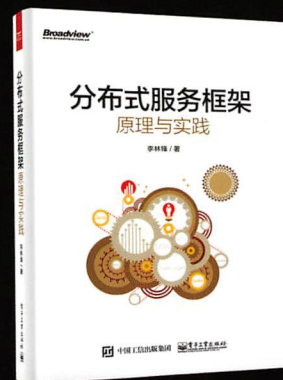
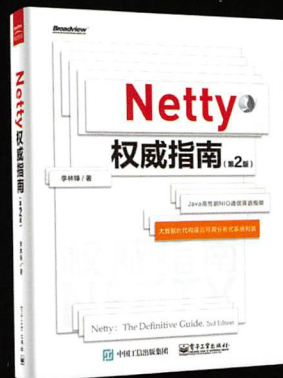
图 20-5 Netty NIO 线程运行堆栈

如果发现性能瓶颈在网络 I/O 读写上，可以适当调大 `NioEventLoopGroup` 中的 `work I/O` 线程数，直到 I/O 处理性能能够满足业务需求。

## 20.3 总结

在实际项目中仅仅会用 Netty 是远远不够的，由于 Netty 承担了 RPC 调用工作，一旦发生问题会导致 RPC 或者微服务调用失败，由此引起的业务中断后果很严重。除了实践，需要熟练掌握 Netty 的核心类库和关键调度流程，这样才能得心应手地解决各种问题。

好/书/分/享



相比传统的阻塞式编程方式，要驾驭Netty的异步编程方式是有一定难度的。Netty为了避免JVM GC和提高性能会直接对内存进行处理，使用者需要对Netty的内存使用有深入了解。本书通过一个个Netty实际使用案例为读者展示了大量Netty技术细节，读者可以快速领悟到Netty专家花大量时间积累的经验。

——华为开源能力中心技术专家、  
红帽软件前首席软件工程师 姜宁

本书带你领略使用Netty过程中的各种“坑”，包括客户端连接池资源泄漏、服务端意外退出、高并发性能波动及IoT海量连接性能问题等。书中除了描述问题的前因后果，还讲解了问题定位的各种思路，“授人以鱼不如授人以渔”，定位问题就像破案，“从蛛丝马迹中寻找线索，从千军万马中取上将首级”，从而快速解决问题。

——华为消费者云服务微服务首席架构师 王世军

Netty已经是网络编程的明星框架了，在阿里Netty也是程序员必须掌握的基础组件。李林锋在本书中总结了工作中遇到的各种问题，通过对问题的深入分析，挖掘Netty框架的工作原理。相信这又是一本非常值得大家深入学习的程序员进阶教材。

——阿里资深技术专家 天民

从2014年开始，李林锋就断断续续地在InfoQ网站上发布了一系列Netty文章，涵盖了Netty技术的方方面面，这些文章深受读者喜欢。这一次他的新书以实践为切入点，从场景出发，为用户深刻地剖析Netty技术的关键点，再加上他酣畅淋漓的文风，看得我甚是陶醉。

——极客邦科技/极客时间总编辑 郭雷

随着云计算、容器、边缘计算、IoT等技术的发展，越来越多的高性能应用构建在轻量级的Netty之上，如API服务网关、IoT设备接入平台、轻量级边缘计算引擎等。在构建过程中我们一定会遇到一些“疑难杂症”，而本书提供了相应的“解药”，可达到“书到病除”之效。

——《亿级流量网站架构核心技术》作者 张开涛

Netty是一款经典的Java开源通信框架，是Java开发工程师必备技能之一。李林锋在高性能通信框架及微服务领域有很深的造诣，本书总结了Netty的各种案例和调优经验，干货满满，不容错过！

——贝壳金服 2B2C CTO、  
微信公众号“IT民工闲话”作者 史海峰

Netty好比Java网络世界中的高铁，坐上它，高性能、高并发网络应用开发似乎变得轻而易举，但李林锋通过这本书告诉我们事实并非如此简单，他用一个个活生生的案例讲述了Netty的“坑”在哪里，“门”又在哪里。

——阿里云弹性计算架构师 蔡俊杰

继《Netty权威指南》之后，非常高兴李林锋又推出了本书，通过一系列Netty应用故障和案例分析，深入Netty内部实现细节，不仅有助于更好地使用Netty，而且可借此了解高并发通信场景下的应用特性，以及Netty的设计思路。技术进阶之路尽在细节与实践。

——《大型网站技术架构：核心原理与案例分析》作者  
李智慧

李林锋是国内Netty技术的先行者和布道者，他写的《Netty权威指南》让开发者认识到了Netty的威力，但是在实际使用Netty时还是会遇到各种问题。本书详细讲解了他在实际使用Netty时遇到的各种问题及分析解决问题的思路和方法，是来自一线宝贵经验总结，对于提高编程水平及分析解决问题的能力大有帮助。

——东软集团架构师、InfoQ社区编辑、  
技术图书译者 张卫滨

认识李林锋已久，一直“神往”，他在微服务架构领域有很深的造诣，经历了从框架层、平台层到业务层的完整过程。本书侧重于程序实战，对Netty开发过程中的难点进行了详述，辅以各种实际业务问题，同时对目前主流的RPC框架进行了剖析，对网络通信开发及RPC选型都有非常好的借鉴意义。

——东方证券首席架构师 樊建

上架建议：程序开发

ISBN 978-7-121-35262-1



9 787121 352621 >

定价：79.00元



博文视点Broadview



@博文视点Broadview



责任编辑：董英  
封面设计：侯士卿 李玲



版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF